A Type-Theoretic Account of Standard ML 1996 (Version 1)

Chris Stone Robert Harper

May 10, 1996
CMU-CS-96-136

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This report also appears as Fox Memorandum CMU-CS-FOX-96-02

This research was sponsored in part by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software", ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050, in part by the National Science Foundation under Grant No. CCR-9502674, and in part by the US Army Research Office under Grant No. DAAH04-94-G-0289. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Advanced Research Projects Agency, the U.S. Government or the National Science Foundation.

Abstract

A type-theoretic definition of a variant of the Standard ML (Revised 1996) programming language is given. The definition consists of a syntax-directed translation of SML96 programs into a typed intermediate language. The intermediate language is an explicitly-typed λ-calculus with product, sum, recursive, and module types. The translation performs type reconstruction, handles identifier scope resolution, enforces static well-formedness conditions, and expands high-level constructs (such as pattern matching and signature matching) into uses of the more rudimentary mechanisms of the intermediate language.

This document presents work in progress and is being distributed for the purpose of obtaining feedback. As such, the translation does not completely match the definition of SML96, which is itself still undergoing change.

Keywords: Standard ML, type theory, semantics, programming language design, compilation, translation, elaboration, static semantics, dynamic semantics

Contents

1 Overview

This document consists of a type-theoretic account of a variant of Standard ML (Revised 1996). The approach taken here is to define the static and dynamic semantics of SML96 by a syntax-directed translation of the SML96 abstract syntax (the *external language (EL)*) into an explicitly-typed λ-calculus (the *internal language (IL)*). The translation performs type inference, resolves the scopes of identifiers, enforces static well-formedness conditions, and renders the high-level constructs of the EL as uses of the more rudimentary mechanisms of the IL. The dynamic semantics of the EL is defined by composition of this translation with the dynamic semantics of the IL itself. This is in contrast to the approach used in *The Definition of Standard ML* [MTH90], which gives a dynamic semantics directly to the "raw" EL independently of the static semantics.

The advantages of this approach are several:

1. The internal language may be shared among many different language definitions given in the style presented here. In particular, we envision the possibility of using the IL given here for the definitions of Scheme and Caml Special Light, perhaps with some relatively minor modifications.

2. The translation given here may be viewed as a reference implementation of a front-end for an SML96 compiler. In particular we envision using this translation as a guide to the extension of the TIL/ML compiler [HM95, Mor95, Tar96, TMC+96] to SML96. Compilers for other languages defined by interpretation into the IL can share the back end of the TIL/ML compiler; only a front-end need be written for each specific language.

3. By defining the language in terms of a translation into an intermediate language, we replace the *ad hoc* static semantic objects of *The Definition of Standard ML* by terms of a well-defined typed λ-calculus. This clarifies a number of issues in the design of SML96. For example, the notion of "type generativity" is replaced by the abstraction mechanisms associated with the formalism of existential types.

There are some disadvantages as well:

1. The translation of datatype declarations is extremely complex. In our view this is a direct reflection of the intrinsic complexity of a mechanism that introduces several mutually recursive abstract types, each a multinary sum of multinary product types.

2. A rigorous definition of the IL must be given in addition to the translation rules from the EL to the IL. This is mitigated somewhat by the possibility of sharing the IL among several language definitions. The burden of defining the IL is shared to some extent by *The Definition*, in the guise of the set of "static semantic objects" and their associated operations.

The type-theoretic definition of SML96 is divided into three main parts.

Type Structure of the Internal Language. The internal language is an explicitly-typed λ-calculus, with a second-class modules system.

The core of the internal language is based on the XML and λ^{ML} calculi [HM93, HMM90]. The constructors of kind Ω (where Ω is the kind of types) include partial and total function types, record types, sum types, reference types, recursive types, and a single extensible sum. Additionally, the constructors are extended with (restricted) tuples of constructors and functions at the constructor level. Note that there are no polymorphic types (polytypes) in our system.

The modules system is based on the translucent sum or manifest type modules calculi [HL94, Ler94]. In addition to translucent signatures, we have total and partial functor signatures. Our subtyping relation on signatures involves only forgetting of type definitions and totality, and not dropping components. This means that subtyping has no run-time effect.

The two levels are connected by the ability to define modules local to a core expression.

Dynamic Semantics of the Internal Language. The dynamic semantics is based on a continuation represented as a "stack" of frames, which in total correspond to an evaluation context in a contextual semantics [WF91]. The handling of references and exceptions is similar to Harper's account of polymorphic references [Har93]. We expect this to be much more amenable to direct mechanization than the style of the *Definition*, since there are no

implicit rules governing the treatment of exceptions. In fact, the semantics comes very close to defining an abstract machine.

Elaboration. The translation is defined by a series of translation judgments of the general form

$$\Gamma \vdash \mathit{EL\text{-}phrase} \rightsquigarrow \mathit{IL\text{-}phrase} : \mathit{IL\text{-}classifier}$$

where Γ is an internal-language context extended with information mapping EL identifiers to IL variables. If Γ^- is the IL context underlying Γ and the above translation judgment holds, we expect the IL typing judgment

$$\Gamma^- \vdash \mathit{IL\text{-}phrase} : \mathit{IL\text{-}classifier}$$

also to hold.

2 Major Differences from SML96

This section refers only to the draft of Standard ML '96 as of the time of this report. Many aspects of SML96 are still under consideration, and may change substantially!

2.1 Equality

This translation does not currently handle equality types, the `eqtype` signature specification, or polymorphic equality. We may relax this restriction (in a later draft of this translation) to allow `eqtype` declarations and the structural equality derivable from these and known types. We do not intend to include equality polymorphism (`''a`), which significantly complicates both the semantics and the implementation of the language.

2.2 Value Restriction

The value restriction used in this translation is different than that currently specified in SML96.

- We treat

  $$\texttt{val}\ \langle\texttt{rec}\rangle\ pat_1\ \texttt{=}\ expr_1\ \texttt{and}\ \ldots\ \texttt{and}\ pat_n\ \texttt{=}\ expr_n$$

  as a derived form for

  $$\texttt{val}\ \langle\texttt{rec}\rangle\ (pat_1, \ldots, pat_n)\ \texttt{=}\ (exp_1, \ldots, exp_n).$$

  If any of $exp_1, \ldots, exp_n$ is not a value, no variables in the entire binding will be generalized. This differs from SSML in which the decision whether to generalize is made on a pattern-by-pattern basis.

- If v is a value, then `val h::t = v` is equivalent to `val h = hd v` and `val t = tl v`. Since in the latter cases `h` and `t` will not be polymorphic (`hd` and `tl` may raise exceptions!), `h` and `t` should not be polymorphic in the former case.

  The translation currently allows variables in a `val` binding to be generalized if the translation of the binding results in a "valuable" structure. At the external language level, this is equivalent to the right-hand side

being a value, and no constants or constructors appearing in the pattern, except for the constructor in a single-constructor datatype. (That is, the pattern must be irrefutable.) Note that tuple-patterns pose no problem, since projection is a total function.

This matches the notion of "polymorphism as substitution," in which `let val` *id*`=`*exp* `in` *exp′* `end` is considered equivalent to $[exp/id]exp'$ when *id* is polymorphic.

- We distinguish constructors from user-defined functions by their types: constructors can have total function types and `fn` expressions should express partial types. However, the translation can sometimes notice that certain user variables and functions are total, and assigns them total types in the translated code. For example,

```
val x = op ::
val y = x (nil, nil)
```

  can give `x` the same total type as the `::` constructor, so `y` can be polymorphic (since a total function applied to a value is valuable and hence generalizable.)

  Thus we allow certain (sound) programs than SML96 would reject; if we added the ability to express total types in the EL, this might even be acceptable. Alternatively, we will have to modify the translation so that it "forgets" that certain values are total functions.

2.3 Type Sharing and Datatypes

The translation allows certain unsatisfiable constraints to appear in EL signatures. In particular, a type sharing constraint equating two "obviously different" datatypes (datatypes with different sets of constructors) will not be flagged. The specification will elaborate, but there will be no structures in the range of the translation that match the specification.

In the language of the *Definition*, we are relaxing the consistency requirement.

2.4 Local and Higher-Order Functors

Our external language permits structure and functor declarations within a `let` or `local` declarations—a reflection of the fact that the internal language has local modules.

2.5 Top-level

We are interested in batch compilers, which may compile top-level declarations separately. Therefore, we require that:

1. Only structures, functors, and signatures may be defined at top-level.
2. All identifiers bound at the top level must be distinct (taking into account the fact that structure, functor, and signature identifier namespaces are always considered disjoint) so that a linker can resolve references correctly.

2.6 Principality

The translation makes no requirement that top-level declarations, or even entire compilation units, have principal environments. A principality requirement may be useful for efficient implementation, and implementations may choose to make such restrictions, but this is purely the decision of the implementor. For example, an implementation could instead delay some typechecking and/or compiling until the entire program becomes known.

2.7 Overloading

This translation does not consider overloaded operators. It may be sufficient to add an "overloaded" type and add a rule to the translation of expressions such that identifiers with overloaded types are coerced to a non-overloaded type whenever it appears. (This would be similar to the current rule that instantiates polymorphic identifiers to monomorphic expressions wherever they appear.)

3 Internal Language Abstract Syntax

3.1 Core Expressions

$$
\begin{array}{rll}
path & ::= & var \mid var.lbls \\
lbls & ::= & lbl \mid lbl.lbls \\
exp & ::= & scon \qquad \text{(constants)} \\
& \mid & var \qquad \text{(variables)} \\
& \mid & loc \qquad \text{(memory locations)} \\
& \mid & name \qquad \text{(tag names for the Any type)} \\
& \mid & exp\ exp' \qquad \text{(application)} \\
& \mid & \mathsf{fix}\ fbnds\ \mathsf{in}\ var\ \mathsf{end} \qquad \text{(projection from fixpoint)} \\
& \mid & \{rbnds\} \qquad \text{(record of values)} \\
& \mid & exp\#lbl \qquad \text{(projection from a record)} \\
& \mid & \mathsf{handle}\ exp\ \mathsf{with}\ exp' \qquad \text{(handle exceptions at type Any)} \\
& \mid & \mathsf{raise}\ exp \qquad \text{(raise exceptions at type Any)} \\
& \mid & \mathsf{catch}\ exp\ \mathsf{with}\ exp' \qquad \text{(handle exceptions at type Unit)} \\
& \mid & \mathsf{fail} \qquad \text{(raise exceptions at type Any)} \\
& \mid & \mathsf{let}\ var{=}mod\ \mathsf{in}\ exp\ \mathsf{end} \qquad \text{(local module definition)} \\
& \mid & \mathsf{new\_stamp}[con] \qquad \text{(new constructor/deconstructor pair)} \\
& \mid & \mathsf{ref}^{con}\ exp \qquad \text{(allocate new ref cell)} \\
& \mid & \mathsf{get}\ exp \qquad \text{(dereference)} \\
& \mid & \mathsf{set}\ exp \qquad \text{(assignment)} \\
& \mid & \mathsf{roll}_i^{con}\ exp \qquad \text{(type coercion into a recursive type)} \\
& \mid & \mathsf{unroll}_i^{con}\ exp \qquad \text{(type coercion out of a recursive type)} \\
& \mid & \mathsf{inj}_i^{con}\ exp \qquad \text{(injection into sum type)} \\
& \mid & \mathsf{proj}_i\ exp \qquad \text{(projection from sum type)} \\
& \mid & \mathsf{tag}^{name}\ exp \qquad \text{(injection into Any type)} \\
& \mid & \mathsf{untag}^{name}\ exp \qquad \text{(projection from Any type)} \\
& \mid & \mathsf{case}^{con}\ (exp_1, \cdots, exp_n)\ \mathsf{of}\ exp\ \mathsf{end} \qquad \text{(case over sum type)} \\
& \mid & mod.lbl \qquad \text{(projection from module)} \\
rbnds & ::= & \cdot \mid rbnds, rbnd \\
rbnd & ::= & lbl \triangleright exp \\
fbnds & ::= & \cdot \mid fbnds, fbnd \\
fbnd & ::= & var'{=}(var{:}con) \mapsto exp
\end{array}
$$

We use "$\lambda\ var{:}con.exp$" as an abbreviation for "$\mathsf{fix}\ var'{=}(var{:}con) \mapsto exp\ \mathsf{in}\ var'\ \mathsf{end}$" where var' does not occur free in exp.

3.2 Core Constructors and Kinds

$con ::=$	var	(type variables)
	$\mid \mathsf{Int} \mid \mathsf{Float} \mid \mathsf{String} \mid \mathsf{Char} \mid \cdots$	(base types)
	$\mid \mathsf{Any}$	(extensible sum type)
	$\mid con\,\mathsf{Ref}$	(reference type)
	$\mid con\,\mathsf{Name}$	(tag-name type)
	$\mid con \rightharpoonup con'$	(partial function type)
	$\mid con \rightarrow con'$	(total function type)
	$\mid con\,[con']$	(application)
	$\mid \mu_i\,[con]$	(projection from fixpoint)
	$\mid \mathsf{Unroll}\,[con]$	(unroll recursive type)
	$\mid \{rdecs\}$	(record type)
	$\mid \lambda\,(var_1, \cdots, var_n).con\ (n \geq 0)$	(constructor-level function)
	$\mid \Sigma\,(con_1, \cdots, con_n)$	(sum type)
	$\mid (con_1, \cdots, con_n)$	(tuple formation)
	$\mid con\#k\ (k \geq 1)$	(tuple projection)
	$\mid mod.lbl$	(module projection)
$rdecs ::=$	$\cdot \mid rdecs, rdec$	
$rdec ::=$	$lbl \triangleright con$	
$knd ::=$	$\Omega^n \quad (n \geq 0)$	(tuples of types)
	$\mid \Omega^m \Rightarrow \Omega^n\ (m, n \geq 0)$	

We refer to the constructor $\{\}$ as Unit.

3.3 Module Expressions

mod ::=	var	
	$[sbnds]$	(structure)
	$(\lambda var{:}sig.mod)$	(functor)
	$mod\ mod'$	(functor application)
	$mod.lbl$	(projection from module)
	$mod{:}sig$	(forgetting type abbreviations)
$sbnds$ ::=	$\cdot$	(structure bindings)
	$sbnds, sbnd$	
$sbnd$ ::=	$lbl \triangleright bnd$	
bnd ::=	$var = exp$	
	$var = mod$	
	$var = con$	

For readability, we often elide the internal names (var's) when writing out $sbnds$ (and $sdecs$). In all cases it should be immediately obvious how to consistently restore variables not appearing anywhere in the inference rule.

3.4 Module Signatures

sig ::=	$[sdecs]$	
	$((var : sig) \rightharpoonup sig')$	(partial functor signature)
	$((var : sig) \rightarrow sig')$	(total functor signature)
$sdecs$ ::=	$\cdot$	(structure declarations)
	$sdecs, sdec$	
$sdec$ ::=	$lbl \triangleright dec$	
$decs$ ::=	$\cdot$	(declaration lists)
	$decs, dec$	
dec ::=	$var{:}con$	
	$var{:}sig$	
	$var{:}knd$	(opaque type declaration)
	$var{:}knd{=}con$	(type abbreviation)
	$loc{:}con$	(typed locations)
	$name{:}con$	(typed exception names)

We use $(sig \rightharpoonup sig')$ and $(sig \rightarrow sig')$ to abbreviate $((var : sig) \rightharpoonup sig')$ and $((var : sig) \rightarrow sig')$ respectively, where var is not free in sig'.

3.5 Top-level

$$
\begin{array}{rcl}
tdecs & ::= & \cdot \\
 & | & tdecs, tdec \\
tdec & ::= & lbl \triangleright var{:}sig \\
 & | & lbl \triangleright var{:}\mathsf{Sig}{=}sig \\
 & & \\
tbnds & ::= & \cdot \\
 & | & tbnds, tbnd \\
tbnd & ::= & lbl \triangleright var{=}mod \\
 & | & lbl \triangleright var{=}sig
\end{array}
$$

3.6 Translation Context

$$\begin{array}{rcl} \Gamma & ::= & \cdot \\ & | & \Gamma, \mathit{lbl} \triangleright \mathit{var} : \mathit{con} \\ & | & \Gamma, \mathit{lbl} \triangleright \mathit{var} : \mathit{knd} \langle = \mathit{con} \rangle \\ & | & \Gamma, \mathit{lbl} \triangleright \mathit{var} : \mathit{sig} \\ & | & \Gamma, \mathit{lbl} \triangleright \mathit{var} : \mathsf{Sig} = \mathit{sig} \end{array}$$

There is an implicit coercion of a translation context (Γ) to a sequence of declarations (*decs*), induced by dropping all signature bindings and forgetting all *lbl*'s.

3.7 Syntactic Restrictions

A constructor (*con*) may only contain at most single total arrow ($\rightarrow$), and if present it must be at the outermost level.

3.8 Notation

- Many grammatical classes (*lbls*, *decs*, *rdecs*, *rbnds*, *sdecs*, *sbnds*, *tdecs*, *tbnds*) specify lists of elements. For each of these classes we define a binary append operation, written with a comma. For example, we define

$$\begin{array}{rcl} \mathit{decs}, \cdot & := & \mathit{decs} \\ \mathit{decs}, (\mathit{decs}', \mathit{dec}') & := & (\mathit{decs}, \mathit{decs}'), \mathit{dec}' \end{array}$$

  with analogous definitions for all the other classes listed above.

- We occasionally use the abbreviation

$$(\mathit{phrase}_i)_{i=1}^{n}$$

  as shorthand for

$$\mathit{phrase}_1, \cdots, \mathit{phrase}_n$$

  (where the phrases are comma-separated).

3.9 Bindings and Scope

We define the functions bound$(\cdot)$ and dom$(\cdot)$ for various declarations (and lists thereof):

Function	*Definition*	
bound(dec)	bound$(var{:}con)$ bound$(var{:}knd)$ bound$(var{:}knd{=}con)$ bound$(var{:}sig)$ bound$(loc{:}con)$ bound$(name{:}con)$	$= var$ $= var$ $= var$ $= var$ $= loc$ $= name$
bound$(decs)$	bound$(dec_1, \cdots, dec_n)$	$= \{\text{bound}(dec_1), \cdots, \text{bound}(dec_n)\}$
bound$(sdecs)$	dom$(lbl_1 \triangleright dec_1, \cdots, lbl_n \triangleright dec_n)$	$= \{\text{bound}(dec_1), \cdots, \text{bound}(dec_n)\}$
bound(bnd)	bound$(var{=}exp)$ bound$(var{=}con)$ bound$(var{=}mod)$	$= var$ $= var$ $= var$
bound$(sbnds)$	dom$(lbl_1 \triangleright bnd_1, \cdots, lbl_n \triangleright bnd_n)$	$= \{\text{bound}(bnd_1), \cdots, \text{bound}(bnd_n)\}$
bound(Γ)	bound$(sdec, \Gamma)$ bound$(lbl \triangleright var{:}\mathsf{Sig}{=}sig, \Gamma)$	$= \{\text{bound}(sdec)\} \cup \text{dom}(\Gamma)$ $= \{var\} \cup \text{dom}(\Gamma)$
dom$(sdecs)$	dom$(lbl_1 \triangleright dec_1, \cdots, lbl_n \triangleright dec_n)$	$= \{lbl_1, \cdots, lbl_n\}$
dom$(rdecs)$	dom$(lbl_1 \triangleright con_1, \cdots, lbl_n \triangleright con_n)$	$= \{lbl_1, \cdots, lbl_n\}$
dom$(tdec)$	dom$(lbl \triangleright var{:}sig)$ dom$(lbl \triangleright var{:}\mathsf{Sig}{=}sig)$	$= lbl$ $= lbl$
dom$(tdecs)$	dom$(tdec_1, \cdots, tdec_n)$	$= \{\text{dom}(tdec_1), \cdots, \text{dom}(tdec_n)\}$
dom(Γ)	dom$(sdec, \Gamma)$ dom$(lbl \triangleright var{:}\mathsf{Sig}{=}sig, \Gamma)$	$= \{\text{dom}(sdec)\} \cup \text{dom}(\Gamma)$ $= \{lbl\} \cup \text{dom}(\Gamma)$

The scopes of bound variables are given by the following table:

Binding Phrase	*Bound*	Scope
fix $fbnds, var'$=(var:con)$\mapsto$$exp, fbnds'$ in var end " let var=mod in exp end	var' var var	entire phrase exp exp
$\lambda\,(var_1, \cdots, var_n).con$	$var_1, \cdots, var_n$	con
$sbnd, sbnds$ $(\lambda var{:}sig.mod)$	bound($sbnd$) var	$sbnds$ mod
$sdec, sdecs$ $(var{:}sig \rightharpoonup sig')$ $(var{:}sig \rightarrow sig')$	bound($sdec$) var var	$sdecs$ sig' sig'
$tbnd, tbnds$ $tdec, tdecs$	bound($tbnd$) bound($tdec$)	$tbnds$ $tdecs$
$sdec, \Gamma$ $lbl \triangleright var$:Sig=sig, Γ	bound($sdec$) var	Γ Γ

We follow standard practice and identify all phrases which differ only with respect to bound variables, locations, and exception names. We use the notation FV(*phrase*) to denote the set of free expression variables in *phrase*, and FTV(*phrase*) to denote the set of free type variables in the phrase.

4 Static Semantics: Typing

4.1 Introduction

In this section we define the well-formedness and typing judgments for the internal language. Points of interest include:

- There are no "semantic objects" in the sense of the *Definition*.
- The rules are explicitly formulated so that a judgment holds only if its constituents (declaration lists, etc.) are well-formed.
- There is a mutual dependency between the typing judgments for (core and module) expressions and the valuability judgments of Section 5.

4.2 Notation

- Optional elements are enclosed by single brackets $\langle \cdots \rangle$.

4.3 Judgment Forms

Judgment...	*Meaning...*
$\vdash decs$ ok	$decs$ is a valid declaration list
$\vdash decs \leq decs'$	$decs$ is a sub-declaration list of $decs'$
$\vdash decs \equiv decs'$	$decs$ and $decs'$ are equivalent declaration lists
$decs \vdash dec$ ok	dec is a valid declaration
$decs \vdash dec \leq dec'$	dec is a sub-declaration of dec'
$decs \vdash dec \equiv dec'$	dec and dec' are equivalent declaration lists
$decs \vdash bnd : dec$	binding bnd has declaration dec
$decs \vdash rdecs$ ok	$rdecs$ is a valid record declaration list
$decs \vdash rdecs \equiv rdecs'$	$rdecs$ and $rdecs'$ are equivalent record declaration lists
$decs \vdash rbnds : rdecs$	record binding list $rbnds$ has declaration list $rdecs$
$decs \vdash knd : \mathsf{Kind}$	knd is a valid kind
$decs \vdash con : knd$	con has kind knd
$decs \vdash con \equiv con' : knd$	con and con' are equivalent constructors of kind knd
$decs \vdash exp : con$	expression exp has type con
$decs \vdash sdecs$ ok	$rdecs$ is a valid signature specification list
$decs \vdash sdecs \leq sdecs'$	$sdecs$ is a sub-specification list of $sdecs'$
$decs \vdash sdecs \equiv sdecs'$	$sdecs$ and $sdecs'$ are equivalent specification lists
$decs \vdash sbnds : sdecs$	function binding list $sbnds$ has declaration list $sdecs$
$decs \vdash sig : \mathsf{Sig}$	sig is a valid signature
$decs \vdash sig \leq sig' : \mathsf{Sig}$	sig is a sub-signature of sig'
$decs \vdash sig \equiv sig' : \mathsf{Sig}$	sig and sig' are equivalent signatures
$decs \vdash mod : sig$	mod has signature sig
$decs \vdash tdecs$ ok	$tdecs$ is a valid top-level declaration list
$decs \vdash tbnds : tdecs$	top-level bindings $tbnds$ have declarations $tdecs$

4.4 Inference Rules

$$\boxed{\vdash decs\ \mathbf{ok}}$$

$$\frac{}{\vdash \cdot\ \mathrm{ok}} \tag{1}$$

$$\frac{decs \vdash dec\ \mathrm{ok}}{\vdash decs, dec\ \mathrm{ok}} \tag{2}$$

Rules 1 and 2: Via the coercion of translation contexts into declaration lists, this judgment induces the judgment $\vdash \Gamma$ ok.

$$\boxed{\vdash decs \leq decs'}$$

$$\frac{}{\vdash \cdot \leq \cdot} \tag{3}$$

$$\frac{\vdash decs \leq decs' \qquad decs \vdash dec \leq dec'}{\vdash decs, dec \leq decs', dec'} \tag{4}$$

$$\boxed{\vdash decs \equiv decs'}$$

$$\frac{}{\vdash \cdot \equiv \cdot} \tag{5}$$

$$\frac{\vdash decs \equiv decs' \qquad decs \vdash dec \equiv dec'}{\vdash decs, dec \equiv decs', dec'} \tag{6}$$

$$\boxed{decs \vdash dec\ \mathbf{ok}}$$

$$\frac{decs \vdash knd : \mathsf{Kind} \qquad var \notin \mathrm{bound}(decs)}{decs \vdash var{:}knd\ \mathrm{ok}} \tag{7}$$

$$\frac{decs \vdash con : knd \qquad var \notin \mathrm{bound}(decs)}{decs \vdash var{:}knd{=}con\ \mathrm{ok}} \tag{8}$$

$$\frac{decs \vdash con : \Omega \qquad var \notin \mathrm{bound}(decs)}{decs \vdash var{:}con\ \mathrm{ok}} \tag{9}$$

$$\frac{decs \vdash sig : \mathsf{Sig} \qquad var \notin \mathrm{bound}(decs)}{decs \vdash var{:}sig\ \mathrm{ok}} \tag{10}$$

$$\frac{decs \vdash con : \Omega \qquad loc \notin \mathrm{bound}(decs)}{decs \vdash loc{:}con\ \mathsf{Ref}\ \mathrm{ok}} \tag{11}$$

$$\frac{decs \vdash con : \Omega \qquad name \notin \mathrm{bound}(decs)}{decs \vdash name{:}con\ \mathsf{Name}\ \mathrm{ok}} \tag{12}$$

$$\boxed{decs \vdash dec \leq dec'}$$

$$\frac{decs \vdash con : knd}{decs \vdash var{:}knd{=}con \leq var{:}knd} \tag{13}$$

$$\frac{decs \vdash sig \leq sig' : \mathsf{Sig}}{decs \vdash var{:}sig \leq var{:}sig'} \tag{14}$$

$$\frac{decs \vdash dec \equiv dec'}{decs \vdash dec \leq dec'} \tag{15}$$

$$\boxed{decs \vdash dec \equiv dec'}$$

$$\frac{decs \vdash knd : \mathsf{Kind}}{decs \vdash var{:}knd \equiv var{:}knd} \tag{16}$$

$$\frac{decs \vdash con \equiv con' : knd}{decs \vdash var{:}knd{=}con \equiv var{:}knd{=}con'} \tag{17}$$

$$\frac{decs \vdash con \equiv con' : \Omega}{decs \vdash var{:}con \equiv var{:}con'} \tag{18}$$

$$\frac{decs \vdash sig \equiv sig' : \mathsf{Sig}}{decs \vdash var{:}sig \equiv var{:}sig'} \tag{19}$$

$$\frac{decs \vdash con \equiv con' : \Omega}{decs \vdash loc{:}con \equiv loc{:}con'} \tag{20}$$

$$\frac{decs \vdash con \equiv con' : \Omega}{decs \vdash name{:}con \equiv name{:}con'} \tag{21}$$

$$\boxed{decs \vdash bnd \ : \ dec}$$

$$\frac{decs \vdash con : knd}{decs \vdash var{=}con \ : \ var{:}knd} \tag{22}$$

$$\frac{decs \vdash con \equiv con' : knd}{decs \vdash var{=}con \ : \ var{:}knd{=}con'} \tag{23}$$

$$\frac{decs \vdash exp : con}{decs \vdash var{=}exp \ : \ var{:}con} \tag{24}$$

$$\frac{decs \vdash mod : sig \qquad decs \vdash sig' \leq sig : \mathsf{Sig}}{decs \vdash var{=}mod \ : \ var{:}sig} \tag{25}$$

$$\boxed{decs \vdash rdecs \ \mathbf{ok}}$$

$$\frac{\vdash decs \ \mathrm{ok}}{decs \vdash \cdot \ \mathrm{ok}} \tag{26}$$

$$\frac{\begin{array}{c} decs \vdash rdecs \ \mathrm{ok} \\ decs \vdash con : \Omega \\ lbl \notin \mathrm{dom}(rdecs) \end{array}}{decs \vdash rdecs, lbl{\triangleright}con \ \mathrm{ok}} \tag{27}$$

$$\boxed{decs \vdash rdecs \ \equiv \ rdecs'}$$

$$\frac{\vdash decs \ \mathrm{ok}}{decs \vdash \cdot \ \equiv \ \cdot} \tag{28}$$

$$\frac{decs \vdash rdecs \ \equiv \ rdecs' \qquad decs \vdash con \equiv con' : \Omega}{decs \vdash rdecs, lbl{\triangleright}con \ \equiv \ rdecs', lbl{\triangleright}con'} \tag{29}$$

$$\frac{decs \vdash rdecs, lbl{\triangleright}con, lbl'{\triangleright}con', rdecs' \ \mathrm{ok}}{decs \vdash rdecs, lbl{\triangleright}con, lbl'{\triangleright}con', rdecs' \ \equiv \ rdecs, lbl'{\triangleright}con', lbl{\triangleright}con, rdecs'} \tag{30}$$

$\boxed{decs \vdash rbnds \ : \ rdecs}$

$$\frac{\vdash decs \text{ ok}}{decs \vdash \cdot \ : \ \cdot} \tag{31}$$

$$\frac{decs \vdash rbnds \ : \ rdecs \qquad decs \vdash exp : con}{decs \vdash rbnds, lbl \triangleright exp \ : \ rdecs, lbl \triangleright con} \tag{32}$$

$\boxed{decs \vdash knd : \mathsf{Kind}}$

$$\frac{n \geq 0}{decs \vdash \Omega^n : \mathsf{Kind}} \tag{33}$$

$$\frac{n, m \geq 0}{decs \vdash \Omega^m \Rightarrow \Omega^n : \mathsf{Kind}} \tag{34}$$

$\boxed{decs \vdash con : knd}$

$$\frac{\begin{array}{c}\vdash decs \text{ ok} \\ decs = decs', var{:}knd\langle{=}con\rangle, decs''\end{array}}{decs \vdash var : knd} \tag{35}$$

$$\frac{}{decs \vdash \mathsf{Any} : \Omega} \tag{36}$$

$$\frac{decs \vdash con : \Omega}{decs \vdash con \, \mathsf{Ref} : \Omega} \tag{37}$$

$$\frac{decs \vdash con : \Omega}{decs \vdash con \, \mathsf{Name} : \Omega} \tag{38}$$

$$\frac{decs \vdash con : \Omega \qquad decs \vdash con' : \Omega}{decs \vdash con \rightharpoonup con' : \Omega} \tag{39}$$

$$\frac{decs \vdash con : \Omega \qquad decs \vdash con' : \Omega}{decs \vdash con \rightarrow con' : \Omega} \tag{40}$$

$$\frac{decs \vdash rdecs \text{ ok}}{decs \vdash \{rdecs\} : \Omega} \tag{41}$$

$$\frac{decs \vdash con : \Omega^n \Rightarrow \Omega^n \qquad 1 \leq i \leq n}{decs \vdash \mu_i \, [con] : \Omega} \tag{42}$$

$$\frac{decs \vdash con \equiv \mu_i\,[con'] : \Omega}{decs \vdash \mathsf{Unroll}\,[con] : \Omega} \tag{43}$$

$$\frac{decs, var_1{:}\Omega, \ldots, var_n{:}\Omega \vdash con : \Omega^m}{decs \vdash \lambda\,(var_1, \ldots, var_n).con : \Omega^n \Rightarrow \Omega^m} \tag{44}$$

$$\frac{decs \vdash con_1 : \Omega \qquad \ldots \qquad decs \vdash con_n : \Omega}{decs \vdash \Sigma\,(con_1, \cdots, con_n) : \Omega} \tag{45}$$

$$\frac{decs \vdash con_1 : \Omega \qquad \ldots \qquad decs \vdash con_n : \Omega}{decs \vdash (con_1, \ldots, con_n) : \Omega^n} \tag{46}$$

$$\frac{decs \vdash con : con' \Rightarrow con \qquad decs \vdash con : knd'}{decs \vdash con\,[con'] : knd} \tag{47}$$

$$\frac{decs \vdash mod : [sdecs, lbl \triangleright var{:}knd\langle{=}con\rangle, sdecs']}{decs \vdash mod.lbl : knd} \tag{48}$$

$$\boxed{decs \vdash con \equiv con' : knd}$$

$$\frac{\begin{array}{c} decs \vdash con \equiv con' : knd \\ decs = decs', var{:}knd{=}con', decs'' \end{array}}{decs \vdash var \equiv con : knd} \tag{49}$$

$$\frac{\begin{array}{c} decs \vdash mod : [lbl_1 \triangleright dec_1, \ldots, lbl_m \triangleright dec_m, lbl \triangleright var{:}knd{=}con', sdecs] \\ decs, dec_1, \ldots, dec_m \vdash con \equiv con' : knd \\ decs \vdash con : knd \end{array}}{decs \vdash mod.lbl \equiv con : knd} \tag{50}$$

Rule 50: The projection must be a valid constructor with respect to the ambient context.

$$\frac{decs \vdash con_1 \equiv con_2 : \Omega \qquad decs \vdash con_1' \equiv con_2' : \Omega}{decs \vdash con_1 \rightharpoonup con_1' \equiv con_2 \rightharpoonup con_2' : \Omega} \tag{51}$$

$$\frac{decs \vdash con_1 \equiv con_2 : \Omega \qquad decs \vdash con_1' \equiv con_2' : \Omega}{decs \vdash con_1 \rightarrow con_1' \equiv con_2 \rightarrow con_2' : \Omega} \tag{52}$$

$$\frac{decs \vdash con \equiv con' : \Omega}{decs \vdash con\,\mathsf{Ref} \equiv con'\,\mathsf{Ref} : \Omega} \tag{53}$$

$$\frac{decs \vdash con \equiv con' : \Omega}{decs \vdash con\,\mathsf{Name} \equiv con'\,\mathsf{Name} : \Omega} \tag{54}$$

$$\frac{decs \vdash rdecs \equiv rdecs'}{decs \vdash \{rdecs\} \equiv \{rdecs'\} : \Omega} \tag{55}$$

$$\frac{decs \vdash con \equiv con' : \Omega^n \Rightarrow \Omega^n \qquad (1 \leq i \leq n)}{decs \vdash \mu_i\,[con] \equiv \mu_i\,[con'] : \Omega} \tag{56}$$

$$\frac{decs \vdash con \equiv \mu_i\,[con'] : \Omega \qquad decs \vdash con' : \Omega^n \Rightarrow \Omega^n}{decs \vdash \mathsf{Unroll}\,[con] \equiv (con'\,[(\mu_1\,[con'], \cdots, \mu_n\,[con'])])\#i : \Omega} \tag{57}$$

$$\frac{decs, var_1{:}\Omega, \ldots, var_n{:}\Omega \vdash con \equiv con' : \Omega}{decs \vdash \lambda\,(var_1, \ldots, var_n).con \equiv \lambda\,(var_1, \ldots, var_n).con' : \Omega^n \Rightarrow \Omega} \tag{58}$$

$$\frac{decs \vdash con_1 \equiv con'_1 : \Omega \qquad \ldots \qquad decs \vdash con_n \equiv con'_n : \Omega}{decs \vdash \Sigma\,(con_1, \cdots, con_n) \equiv \Sigma\,(con'_1, \cdots, con'_n) : \Omega} \tag{59}$$

$$\frac{decs \vdash con_1 \equiv con'_1 : \Omega \qquad \ldots \qquad decs \vdash con_n \equiv con'_n : \Omega}{decs \vdash (con_1, \ldots, con_n) \equiv (con'_1, \cdots, con'_n) : \Omega^n} \tag{60}$$

$$\frac{decs \vdash con_1 \equiv con'_1 : knd' \Rightarrow knd \qquad decs \vdash con_2 \equiv con'_2 : knd'}{decs \vdash con_1\,[con_2] \equiv con'_1\,[con'_2] : knd} \tag{61}$$

$$\frac{\begin{array}{c} decs, var_1{:}\Omega, \ldots, var_n{:}\Omega \vdash con : \Omega^m \\ decs \vdash con' \equiv (con'_1, \ldots, con'_n) : \Omega^n \end{array}}{decs \vdash (\lambda\,(var_1, \ldots, var_n).con)\,[con'] \equiv [con'_1/var_1] \cdots [con'_n/var_n]con : \Omega^m} \tag{62}$$

$$\frac{decs \vdash con : \Omega^n \Rightarrow \Omega^m}{decs \vdash \lambda\,(var_1, \ldots, var_n).(con\,[(var_1, \ldots, var_n)]) \equiv con : \Omega^m} \tag{63}$$

$$\frac{decs \vdash con \equiv (con_1, \ldots, con_n) : \Omega^n \qquad (1 \leq i \leq n)}{decs \vdash con\#i \equiv con_i : \Omega} \tag{64}$$

$$\frac{decs \vdash con : \Omega^n}{decs \vdash con \equiv (con\#1, \ldots, con\#n) : \Omega^n} \tag{65}$$

$$\frac{decs \vdash con : knd}{decs \vdash con \equiv con : knd} \tag{66}$$

$$\frac{decs \vdash con' \equiv con : knd}{decs \vdash con \equiv con' : knd} \tag{67}$$

$$\frac{decs \vdash con \equiv con' : knd \qquad decs \vdash con' \equiv con'' : knd}{decs \vdash con \equiv con'' : knd} \tag{68}$$

$$\boxed{decs \vdash exp : con}$$

$$\frac{\begin{array}{c}\vdash decs \text{ ok} \\ decs = decs', var{:}con, decs''\end{array}}{decs \vdash var : con} \tag{69}$$

$$\frac{\begin{array}{c}\vdash decs \text{ ok} \\ decs = decs', loc{:}con, decs''\end{array}}{decs \vdash loc : con} \tag{70}$$

$$\frac{\begin{array}{c}\vdash decs \text{ ok} \\ decs = decs', name{:}con, decs''\end{array}}{decs \vdash name : con} \tag{71}$$

$$\frac{decs \vdash exp : con' \rightharpoonup con \qquad decs \vdash exp' : con'}{decs \vdash exp\ exp' : con} \tag{72}$$

$$\frac{decs \vdash exp : con' \to con \qquad decs \vdash exp' : con'}{decs \vdash exp\ exp' : con} \tag{73}$$

$$\frac{decs, (var'_i{:}con_i \rightharpoonup con'_i)_{i=1}^{n}, var_k{:}con_k \vdash exp_k : con'_k \qquad (1 \le k \le n)}{decs \vdash \mathsf{fix}\,(var'_i{=}(var_i{:}con_i) \mapsto exp_i)_{i=1}^{n}\ \mathsf{in}\ var'_k\ \mathsf{end} : con_k \rightharpoonup con'_k} \tag{74}$$

$$\frac{\begin{array}{c}var'_1, \cdots, var'_n \notin \mathrm{bound}(decs) \\ decs, var_k{:}con_k \vdash exp_k \downarrow con'_k\end{array}}{decs \vdash \mathsf{fix}\,(var'_i{=}(var_i{:}con_i) \mapsto exp_i)_{i=1}^{n}\ \mathsf{in}\ var'_k\ \mathsf{end} : con_k \to con'_k} \tag{75}$$

Rule 75: This rule is actually overly restricts the functions that can be given total types, but it suffices.

$$\frac{decs \vdash rbnds \ : \ rdecs}{decs \vdash \{rbnds\} : \{rdecs\}} \tag{76}$$

$$\frac{decs \vdash exp : \{rdecs, lbl \triangleright con, rdecs'\}}{decs \vdash exp \# lbl : con} \tag{77}$$

$$\frac{decs \vdash exp : con \qquad decs \vdash exp' : \mathsf{Any} \rightharpoonup con}{decs \vdash \mathsf{handle}\ exp\ \mathsf{with}\ exp' : con} \tag{78}$$

$$\frac{decs \vdash exp : \mathsf{Any}}{decs \vdash \mathsf{raise}\ exp : con} \tag{79}$$

$$\frac{decs \vdash exp : con \qquad decs \vdash exp' : con}{decs \vdash \mathsf{catch}\ exp\ \mathsf{with}\ exp' : con} \tag{80}$$

$$\frac{}{decs \vdash \mathsf{fail} : con} \tag{81}$$

$$\frac{decs \vdash mod : sig \qquad decs, var{:}sig \vdash exp : con \qquad decs \vdash con : \Omega}{decs \vdash \mathsf{let}\ var{=}mod\ \mathsf{in}\ exp\ \mathsf{end} : con} \tag{82}$$

Rule 82: The result type may not depend on abstract types from the local module.

$$\frac{}{decs \vdash \mathsf{new\_stamp}[con] : \{\mathrm{mk}{:}con \rightarrow \mathsf{Any}, \mathrm{km}{:}\mathsf{Any} \rightharpoonup con\}} \tag{83}$$

$$\frac{decs \vdash exp : con}{decs \vdash \mathsf{ref}^{con}\ exp : con\ \mathsf{Ref}} \tag{84}$$

$$\frac{decs \vdash exp : con\ \mathsf{Ref}}{decs \vdash \mathsf{get}\ exp : con} \tag{85}$$

$$\frac{decs \vdash exp : \{1 \triangleright con\ \mathsf{Ref}, 2 \triangleright con\}}{decs \vdash \mathsf{set}\ exp : \{\}} \tag{86}$$

$$\frac{\begin{array}{c} decs \vdash con' \equiv \mu_i\,[con] : \Omega \\ decs \vdash con : \Omega^n \Rightarrow \Omega^n \qquad 1 \leq i \leq n \\ decs \vdash exp : \mathsf{Unroll}\,[\mu_i\,[con]] \end{array}}{decs \vdash \mathsf{roll}_i^{con'}\ exp : \mu_i\,[con]} \tag{87}$$

$$\frac{\begin{array}{c} decs \vdash con' \equiv \mu_i\,[con] : \Omega \\ decs \vdash con : \Omega^n \Rightarrow \Omega^n \qquad 1 \leq i \leq n \\ decs \vdash exp : \mu_i\,[con] \end{array}}{decs \vdash \mathsf{unroll}_i^{con'}\ exp : \mathsf{Unroll}\,[\mu_i\,[con]]} \tag{88}$$

$$\frac{\begin{array}{c} decs \vdash con \equiv \Sigma\,(con_1, \cdots, con_n) : \Omega \\ 1 \leq i \leq n \qquad decs \vdash exp : con_i \end{array}}{decs \vdash \mathsf{inj}_i^{con}\ exp : con} \tag{89}$$

$$\frac{decs \vdash exp : \Sigma\,(con_1, \cdots, con_n) \qquad 1 \leq i \leq n}{decs \vdash \mathsf{proj}_i\ exp : con} \tag{90}$$

$$\frac{decs \vdash name : con\,\mathsf{Name} \qquad decs \vdash exp : con}{decs \vdash \mathsf{tag}^{name}\ exp : \mathsf{Any}} \tag{91}$$

$$\frac{decs \vdash name : con\,\mathsf{Name} \qquad decs \vdash exp : \mathsf{Any}}{decs \vdash \mathsf{untag}^{name}\ exp : con} \tag{92}$$

$$\frac{\begin{array}{c} decs \vdash con \equiv \Sigma\,(con_1, \cdots, con_n) : \Omega \\ decs \vdash exp : con \\ decs \vdash exp_1 : con \rightharpoonup con' \qquad \ldots \qquad decs \vdash exp_n : con \rightharpoonup con' \end{array}}{decs \vdash \mathsf{case}^{con}\ (exp_1, \cdots, exp_n)\ \mathsf{of}\ exp\ \mathsf{end} : con'} \tag{93}$$

Rule 93: Note that the case analysis does not do any destructuring of the sum value.

$$\frac{\begin{array}{c} decs \vdash mod : [lbl_1 \triangleright dec_1, \ldots, lbl_m \triangleright dec_m, lbl \triangleright var{:}con', sdecs] \\ decs, dec_1, \ldots, dec_m \vdash con \equiv con' : \Omega \\ decs \vdash con : \Omega \end{array}}{decs \vdash mod.lbl : con} \tag{94}$$

Rule 94: A projection is only well-formed if the result is typable. If *mod* is valuable, the projection is always well-formed.

$$\frac{decs \vdash exp : con \rightarrow con'}{decs \vdash exp : con \rightharpoonup con'} \tag{95}$$

Rule 95: Since we only allow a total arrow at the outermost level of a constructor, this rule suffices to completely describe the "subtyping" of constructors.

$$\frac{decs \vdash exp : con' \qquad decs \vdash con \equiv con' : \Omega}{decs \vdash exp : con} \tag{96}$$

Rule 96: We could remove this non-syntax-directed rule by adding explicit constructor-equivalence tests to many of the the above rules.

$$\boxed{decs \vdash sdecs\ \mathbf{ok}}$$

$$\frac{\vdash decs\ \mathrm{ok}}{decs \vdash \cdot\ \mathrm{ok}} \tag{97}$$

$$\frac{\begin{array}{c} decs \vdash dec\ \mathrm{ok} \\ decs, dec \vdash sdecs\ \mathrm{ok} \\ lbl \notin \mathrm{dom}(sdecs) \end{array}}{decs \vdash lbl \triangleright dec, sdecs\ \mathrm{ok}} \tag{98}$$

$$\boxed{decs \vdash sdecs \leq sdecs'}$$

$$\frac{}{decs \vdash \cdot \leq \cdot} \tag{99}$$

$$\frac{dec \vdash dec \leq dec' \qquad decs, dec \vdash sdecs \leq sdecs'}{decs \vdash lbl \triangleright dec, sdecs \leq lbl \triangleright dec', sdecs'} \tag{100}$$

$$\boxed{decs \vdash sdecs \equiv sdecs'}$$

$$\frac{}{decs \vdash \cdot \equiv \cdot} \tag{101}$$

$$\frac{dec \vdash dec \equiv dec' \qquad decs, dec \vdash sdecs \equiv sdecs'}{decs \vdash lbl \triangleright dec, sdecs \equiv lbl \triangleright dec', sdecs'} \tag{102}$$

$$\frac{\begin{array}{c} decs \vdash sdecs, lbl \triangleright dec, lbl' \triangleright dec', sdecs'\ \mathrm{ok} \\ \mathrm{bound}(dec') \cap (\mathrm{FV}(dec) \cup \mathrm{FTV}(dec)) = \emptyset \\ \mathrm{bound}(dec) \cap (\mathrm{FV}(dec') \cup \mathrm{FTV}(dec')) \end{array}}{decs \vdash sdecs, lbl \triangleright dec, lbl' \triangleright dec', sdecs' \equiv sdecs, lbl' \triangleright dec', lbl \triangleright dec, sdecs'} \tag{103}$$

$$\boxed{decs \vdash sbnds \ : \ sdecs}$$

$$\frac{}{decs \vdash \cdot \ : \ \cdot} \tag{104}$$

$$\frac{decs \vdash bnd \ : \ dec \qquad decs, dec \vdash sbnds \ : \ sdecs}{decs \vdash lbl \triangleright bnd, sbnds \ : \ lbl \triangleright dec, sdecs} \tag{105}$$

$$\boxed{decs \vdash sig : \mathsf{Sig}}$$

$$\frac{decs \vdash sdecs \text{ ok}}{decs \vdash [sdecs] : \mathsf{Sig}} \tag{106}$$

$$\frac{decs, var{:}sig \vdash sig' : \mathsf{Sig}}{decs \vdash (var{:}sig \rightharpoonup sig') : \mathsf{Sig}} \tag{107}$$

$$\frac{decs, var{:}sig \vdash sig' : \mathsf{Sig}}{decs \vdash (var{:}sig \to sig') : \mathsf{Sig}} \tag{108}$$

$$\boxed{decs \vdash sig \leq sig' : \mathsf{Sig}}$$

$$\frac{decs \vdash sdecs \ \leq \ sdecs'}{decs \vdash [sdecs] \leq [sdecs'] : \mathsf{Sig}} \tag{109}$$

$$\frac{decs \vdash sig_2 \leq sig_1 : \mathsf{Sig} \qquad decs, var{:}sig_2 \vdash sig_1' \leq sig_2' : \mathsf{Sig}}{decs \vdash (var{:}sig_1 \rightharpoonup sig_1') \leq (var{:}sig_2 \rightharpoonup sig_2') : \mathsf{Sig}} \tag{110}$$

$$\frac{decs \vdash sig_2 \leq sig_1 : \mathsf{Sig} \qquad decs, var{:}sig_2 \vdash sig_1' \leq sig_2' : \mathsf{Sig}}{decs \vdash (var{:}sig_1 \to sig_1') \leq (var{:}sig_2 \rightharpoonup sig_2') : \mathsf{Sig}} \tag{111}$$

$$\frac{decs \vdash sig_2 \leq sig_1 : \mathsf{Sig} \qquad decs, var{:}sig_2 \vdash sig_1' \leq sig_2' : \mathsf{Sig}}{decs \vdash (var{:}sig_1 \to sig_1') \leq (var{:}sig_2 \to sig_2') : \mathsf{Sig}} \tag{112}$$

$$\boxed{decs \vdash sig \equiv sig' : \mathsf{Sig}}$$

$$\frac{decs \vdash sdecs \equiv sdecs'}{decs \vdash [sdecs] \equiv [sdecs'] : \mathsf{Sig}} \tag{113}$$

$$\frac{decs \vdash sig_1 \equiv sig_2 : \mathsf{Sig} \qquad decs, var{:}sig_1 \vdash sig_1' \equiv sig_2' : \mathsf{Sig}}{decs \vdash (var{:}sig_1 \rightharpoonup sig_1') \equiv (var{:}sig_2 \rightharpoonup sig_2') : \mathsf{Sig}} \tag{114}$$

$$\frac{decs \vdash sig_1 \equiv sig_2 : \mathsf{Sig} \qquad decs, var{:}sig_1 \vdash sig_1' \equiv sig_2' : \mathsf{Sig}}{decs \vdash (var{:}sig_1 \rightarrow sig_1') \equiv (var{:}sig_2 \rightarrow sig_2') : \mathsf{Sig}} \tag{115}$$

$$\boxed{decs \vdash mod : sig}$$

$$\frac{\vdash decs \text{ ok} \qquad decs = decs', var{:}sig, decs''}{decs \vdash var : sig} \tag{116}$$

$$\frac{decs \vdash sbnds : sdecs}{decs \vdash [sbnds] : [sdecs]} \tag{117}$$

$$\frac{decs, var{:}sig \vdash mod : sig'}{decs \vdash (\lambda var{:}sig.mod) : (var{:}sig \rightharpoonup sig')} \tag{118}$$

$$\frac{decs, var{:}sig \vdash mod \downarrow sig'}{decs \vdash (\lambda var{:}sig.mod) : (var{:}sig \rightarrow sig')} \tag{119}$$

$$\frac{decs \vdash mod : (sig' \rightharpoonup sig) \qquad decs \vdash mod' : sig'}{decs \vdash mod\, mod' : sig} \tag{120}$$

Rule 120: Only functors with non-dependent types may be applied. Dependencies can be eliminated by uses of the subtyping and equivalence rules. If the argument is valuable, dependencies can always be eliminated.

$$\frac{\begin{array}{c} decs \vdash mod : [lbl_1 \triangleright dec_1, \ldots, lbl_m \triangleright dec_m, lbl \triangleright var{:}sig', sdecs] \\ decs, dec_1, \ldots, dec_m \vdash sig \equiv sig' : \mathsf{Sig} \\ decs \vdash sig : \mathsf{Sig} \end{array}}{decs \vdash mod.lbl : sig} \tag{121}$$

Rule 121: A projection is only well-formed if the result can be given a signature in the ambient context. If *mod* is valuable, the projection is always well-formed.

$$\frac{decs \vdash mod : sig}{decs \vdash mod{:}sig : sig} \tag{122}$$

Rule 122: Ascription of a signature to a module can permanently forget type abbreviations.

$$\frac{decs \vdash mod \downarrow [sdecs, lbl \triangleright var{:}knd, sdecs']}{decs \vdash mod : [sdecs, lbl \triangleright var{:}knd{=}mod.lbl, sdecs']} \tag{123}$$

Rule 123: The "self" rule. If *mod.lbls* specifies a type and *mod* has a well-defined value (is valuable) then $mod.lbls \equiv mod.lbls$; we add this fact to the signature.

$$\frac{decs \vdash mod \downarrow [sdecs, lbl \triangleright var{:}sig, sdecs'] \qquad decs \vdash mod.lbl : sig'}{decs \vdash mod : [sdecs, lbl \triangleright var{:}sig', sdecs']} \tag{124}$$

Rule 124: Allows the "self" rule to be applied to substructures.

$$\frac{decs \vdash mod : sig' \qquad decs \vdash sig' \leq sig : \mathsf{Sig}}{decs \vdash mod : sig} \tag{125}$$

$$\boxed{decs \vdash tdecs\ \mathbf{ok}}$$

$$\frac{\vdash decs\ \mathrm{ok}}{decs \vdash \cdot\ \mathrm{ok}} \tag{126}$$

$$\frac{decs \vdash sig : \mathsf{Sig} \qquad decs, var{:}sig \vdash tdecs\ \mathrm{ok} \qquad lbl \notin \mathrm{dom}(tdecs)}{decs \vdash lbl \triangleright var{:}sig, tdecs\ \mathrm{ok}} \tag{127}$$

$$\frac{decs \vdash sig : \mathsf{Sig} \qquad decs \vdash tdecs\ \mathrm{ok} \qquad lbl \notin \mathrm{dom}(tdecs)}{decs \vdash lbl \triangleright var{:}\mathsf{Sig}{=}sig, tdecs\ \mathrm{ok}} \tag{128}$$

$$\boxed{decs \vdash tbnds \ : \ tdecs}$$

$$\frac{decs \vdash \ \mathrm{ok}}{decs \vdash \cdot \ : \ \cdot} \tag{129}$$

$$\frac{decs \vdash mod : sig \qquad decs, var{:}sig \vdash tbnds \ : \ tdecs}{decs \vdash \ lbl \triangleright var{=}mod, tbnds \ : \ lbl \triangleright var{:}sig, tdecs} \tag{130}$$

$$\frac{decs \vdash sig \equiv sig' : \mathsf{Sig} \qquad decs \vdash tbnds \ : \ tdecs \qquad var \notin \mathrm{bound}(decs)}{decs \vdash \ lbl \triangleright var{=}sig, tbnds \ : \ lbl \triangleright var{:}\mathsf{Sig}{=}sig', tdecs} \tag{131}$$

5 Static Semantics: Valuability

The set of *valuable* expressions is a class of expression which evaluate to a value without side-effects, referencing the store, or raising exceptions. Similarly, valuable modules are modules which evaluate without side-effects, referencing the store, or raising exceptions. The purpose of distinguishing total and partial functions, as well as total and partial functors, is to specify which function/functor applications are valuable.

In the translation, only EL expressions whose translation is valuable will be polymorphically generalized. This means that general function applications will not be generalized, but constructor applications may be (since constructors have total arrows while general functions have partial arrows).

5.1 Judgment Forms

Judgment...	*Meaning...*
$decs \vdash exp \downarrow con$	exp is a valuable expression of type *con*
$decs \vdash mod \downarrow sig$	*mod* is a valuable module with signature *sig*
$decs \vdash exp \downarrow$	exp is a valuable expression
$decs \vdash mod \downarrow$	*mod* is a valuable module

5.2 Syntactic Values

We view each class of syntactic values ($class_v$) as subsets of the corresponding class in the abstract syntax ($class$).

$$
\begin{array}{rcl}
exp_{\mathrm{v}} & ::= & scon \\
& | & \{rbnds_{\mathrm{v}}\} \\
& | & \mathsf{fix}\ fbnds\ \mathsf{in}\ var\ \mathsf{end} \\
& | & \mathsf{inj}_i^{con}\ exp_{\mathrm{v}} \\
& | & \mathsf{tag}^{name}\ exp_{\mathrm{v}} \\
\\
rbnds_{\mathrm{v}} & ::= & \cdot \\
& | & rbnds_{\mathrm{v}}, rbnd_{\mathrm{v}} \\
rbnd_{\mathrm{v}} & ::= & lbl \triangleright exp_{\mathrm{v}} \\
\\
mod_{\mathrm{v}} & ::= & [sbnds_{\mathrm{v}}] \\
& | & (\lambda var{:}sig.mod) \\
\\
bnd_{\mathrm{v}} & ::= & var{=}exp_{\mathrm{v}} \\
& | & var{=}mod_{\mathrm{v}} \\
& | & var{=}con \\
\\
sbnds_{\mathrm{v}} & ::= & \cdot \\
& | & sbnds_{\mathrm{v}}, sbnd_{\mathrm{v}} \\
sbnd_{\mathrm{v}} & ::= & lbl \triangleright bnd_{\mathrm{v}} \\
\\
val & ::= & exp_{\mathrm{v}} \\
& | & mod_{\mathrm{v}} \\
& | & con
\end{array}
$$

5.3 Inference Rules

$$\boxed{decs \vdash exp \downarrow}$$

$$\frac{}{decs \vdash exp_{\mathrm{v}} \downarrow} \tag{132}$$

$$\frac{decs \vdash mod \downarrow}{decs \vdash mod.lbl \downarrow} \tag{133}$$

$$\frac{decs \vdash exp_1 \downarrow con' \rightarrow con \qquad decs \vdash exp_2 \downarrow}{decs \vdash exp_1\ exp_2 \downarrow} \tag{134}$$

$$\frac{decs \vdash exp_1 \downarrow \qquad \cdots \qquad decs \vdash exp_n \downarrow}{decs \vdash \{lbl_1 \triangleright exp_1, \cdots, lbl_n \triangleright exp_n\} \downarrow} \tag{135}$$

$$\frac{decs \vdash mod \downarrow sig \qquad decs, var{:}sig \vdash exp \downarrow}{decs \vdash \mathsf{let}\ var{=}mod\ \mathsf{in}\ exp\ \mathsf{end} \downarrow} \tag{136}$$

$$\frac{decs \vdash exp \downarrow}{decs \vdash \mathsf{roll}_i^{con}\ exp \downarrow} \tag{137}$$

$$\frac{decs \vdash exp \downarrow}{decs \vdash \mathsf{unroll}_i^{con}\ exp \downarrow} \tag{138}$$

$$\frac{decs \vdash exp \downarrow \Sigma\,(con)}{decs \vdash \mathsf{proj}_1\ exp \downarrow} \tag{139}$$

Rule 139: Projection from a single-element sum type is total.

$$\boxed{decs \vdash sbnds \downarrow}$$

$$\frac{}{decs \vdash \cdot \downarrow} \tag{140}$$

$$\frac{decs \vdash exp \downarrow con \qquad decs, var{:}consbnds \vdash \downarrow}{decs \vdash lbl \triangleright var{=}exp, sbnds \downarrow} \tag{141}$$

$$\frac{decs, var{:}knd\langle{=}con\rangle \vdash sbnds \downarrow}{decs \vdash lbl \triangleright var{=}knd\langle{=}con\rangle, sbnds \downarrow} \tag{142}$$

$$\frac{decs \vdash mod \downarrow sig \qquad decs, var{:}sig \vdash sbnds \downarrow}{decs \vdash lbl \triangleright var{=}mod, sbnds \downarrow} \tag{143}$$

$$\boxed{decs \vdash mod \downarrow}$$

$$\frac{}{decs \vdash [sbnds] \downarrow} \tag{144}$$

$$\frac{}{decs \vdash var \downarrow} \tag{145}$$

$$\frac{decs \vdash mod \downarrow (sig' \rightarrow sig) \qquad decs \vdash mod' \downarrow}{decs \vdash mod\ mod' \downarrow} \tag{146}$$

$$\frac{decs \vdash mod \downarrow}{decs \vdash mod.lbl \downarrow} \tag{147}$$

$$\boxed{decs \vdash exp \downarrow con}$$

$$\frac{decs \vdash exp : con \qquad decs \vdash exp \downarrow}{decs \vdash exp \downarrow con} \tag{148}$$

$$\boxed{decs \vdash mod \downarrow sig}$$

$$\frac{decs \vdash mod : sig \qquad decs \vdash mod \downarrow}{decs \vdash mod \downarrow sig} \tag{149}$$

6 Dynamic Semantics

6.1 Introduction

The dynamic semantics is written as a natural semantics, but could (and probably should) easily be turned into the specification of an abstract machine.

The components of the judgments include:

- Δ, a context (*decs*) giving the types of locations and exception names. This is used to choose "new" locations and exception names, and to allow a proof of type-preservation.
- σ, a store mapping the locations in bound(Δ) to values.
- E, a "stack" of frames which all together represent a continuation or evaluation context. If F ranges over frames (see Section 6.2), then we define the grammar for E by

$$\begin{array}{rcl} E & ::= & [] \\ & | & E \circ F \end{array}$$

  As a technical convenience, we let R range over a subset of frames called "raise frames." These are the frames which propagate all exceptions.

- *ans*, an "answer" resulting from evaluating a core or module expression, with the following grammar:

$$\begin{array}{rcll} ans & ::= & \mathsf{VALUE}(val) & \text{(Result is a value)} \\ & | & \mathsf{UNCAUGHT}(exp_{\mathrm{v}}) & \text{(Uncaught exception)} \\ & | & \mathsf{FAIL} & \text{(Uncaught failure)} \end{array}$$

- *Ans*, an "answer" resulting from evaluating an entire program (*tbnds*), with the following grammar:

$$\begin{array}{rcll} Ans & ::= & \mathsf{VALUES}(tbnds) & \text{(Program result)} \\ & | & \mathsf{UNCAUGHT}(exp_{\mathrm{v}}) & \text{(Uncaught exception)} \\ & | & \mathsf{FAIL} & \text{(Uncaught failure)} \end{array}$$

6.2 Frames

F ::=

- $[]\ exp$
- $exp_{\mathrm{v}}\ []$
- $\{rbnds_{\mathrm{v}}, lbl \triangleright [], rbnds\}$
- $[]\#lbl$
- handle $[]$ with exp
- raise $[]$
- catch $[]$ with exp
- let $var = []$ in exp end
- $\mathsf{ref}^{con}\ []$
- get $[]$
- set $[]$
- $\mathsf{roll}_i^{con}\ []$
- $\mathsf{unroll}_i^{con}\ []$
- $\mathsf{inj}_i^{con}\ []$
- $\mathsf{proj}_i\ []$
- $\mathsf{tag}^{name}\ []$
- $\mathsf{untag}^{name}\ []$
- $\mathsf{case}^{con}\ (exp_1, \cdots, exp_n)$ of $[]$ end
- $[sbnds_{\mathrm{v}}, lbl \triangleright var = [], sbnds]$
- $[]\ mod$
- $mod_{\mathrm{v}}\ []$
- $[].lbl$
- $[]{:}sig$

6.3 Raise Frames

$$
\begin{array}{rl}
R ::= & [] \\
| & [] \; exp \\
| & exp_{\mathrm{v}} \; [] \\
| & \{rbnds_{\mathrm{v}}, lbl \triangleright [], rbnds\} \\
| & []\#lbl \\
| & \mathsf{raise}\,[] \\
| & \mathsf{let}\; var = [] \;\mathsf{in}\; exp \;\mathsf{end} \\
| & \mathsf{ref}^{con}\; [] \\
| & \mathsf{get}\; [] \\
| & \mathsf{set}\; [] \\
| & \mathsf{roll}_i^{con}\; [] \\
| & \mathsf{unroll}_i^{con}\; [] \\
| & \mathsf{inj}_i^{con}\; [] \\
| & \mathsf{proj}_i\; [] \\
| & \mathsf{tag}^{name}\; [] \\
| & \mathsf{untag}^{name}\; [] \\
| & \mathsf{case}^{con}\; (exp_1, \cdots, exp_n)\; \mathsf{of}\; []\; \mathsf{end} \\
| & [sbnds_{\mathrm{v}}, lbl \triangleright var = [], sbnds] \\
| & [] \; mod \\
| & mod_{\mathrm{v}}\; [] \\
| & [].lbl \\
| & []{:}sig
\end{array}
$$

6.4 Judgment Forms

Judgment...	*Meaning...*
$\Delta, \sigma, E \vdash exp \Downarrow \Delta', \sigma', ans$	Evaluation of expression
$\Delta, \sigma, E \vdash mod \Downarrow \Delta', \sigma', ans$	Evaluation of module expression
$val \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans$	Plug value back into most recent frame
$\mathsf{raise}\ exp_{\mathrm{v}} \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans$	Propagate exception
$\mathsf{fail} \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans$	Propagate failure
$\Delta, \sigma \vdash tbnds \Downarrow \Delta', \sigma', Ans$	Evaluation of entire program

6.5 Inference Rules

$$\boxed{\Delta, \sigma, E \vdash exp \Downarrow \Delta', \sigma', ans}$$

$$\frac{val \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash val \Downarrow \Delta', \sigma', ans} \tag{150}$$

$$\frac{\Delta, \sigma, E \circ [[]\ exp'] \vdash exp \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash exp\ exp' \Downarrow \Delta', \sigma', ans} \tag{151}$$

$$\frac{\Delta, \sigma, E \circ [\{lbl \triangleright [], rbnds\}] \vdash exp \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash \{lbl \triangleright exp, rbnds\} \Downarrow \Delta', \sigma', ans} \tag{152}$$

Rule 152: A record value can evaluate to itself in one step (via Rule 150) or here component-by-component.

$$\frac{\Delta, \sigma, E \circ [[]\#lbl] \vdash exp \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash exp\#lbl \Downarrow \Delta', \sigma', ans} \tag{153}$$

$$\frac{\Delta, \sigma, E \circ [\mathsf{handle}\ []\ \mathsf{with}\ exp'] \vdash exp \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash \mathsf{handle}\ exp\ \mathsf{with}\ exp' \Downarrow \Delta', \sigma', ans} \tag{154}$$

$$\frac{\Delta, \sigma, E \circ [\mathsf{raise}\ []] \vdash exp \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash \mathsf{raise}\ exp \Downarrow \Delta', \sigma', ans} \tag{155}$$

$$\frac{\Delta, \sigma, E \circ [\mathsf{catch}\ []\ \mathsf{with}\ exp'] \vdash exp \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash \mathsf{catch}\ exp\ \mathsf{with}\ exp' \Downarrow \Delta', \sigma', ans} \tag{156}$$

$$\frac{\mathsf{fail} \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash \mathsf{fail} \Downarrow \Delta', \sigma', ans} \tag{157}$$

$$\frac{\Delta, \sigma, E \circ [\mathsf{let}\ var{=}[]\ \mathsf{in}\ exp\ \mathsf{end}] \vdash mod \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash \mathsf{let}\ var{=}mod\ \mathsf{in}\ exp\ \mathsf{end} \Downarrow \Delta', \sigma', ans} \tag{158}$$

$$\frac{\begin{array}{c} name \notin \mathrm{dom}(\Delta) \\ \{\mathrm{mk}{\triangleright}\lambda\, var{:}con.\mathsf{tag}^{name}\ var, \\ \ \mathrm{km}{\triangleright}\lambda\, var{:}\mathsf{Any.untag}^{name}\ var\} \vdash \Delta[name{:}con\ \mathsf{Name}], \sigma, E \Downarrow \Delta', \sigma', ans \end{array}}{\Delta, \sigma, E \vdash \mathsf{new\_stamp}[con] \Downarrow \Delta', \sigma', ans} \tag{159}$$

$$\frac{\Delta, \sigma, E \circ [\mathsf{ref}^{con}\ []] \vdash exp \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash \mathsf{ref}^{con}\ exp \Downarrow \Delta', \sigma', ans} \tag{160}$$

$$\frac{\Delta, \sigma, E \circ [\mathsf{get}\ []] \vdash exp \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash \mathsf{get}\ exp \Downarrow \Delta', \sigma', ans} \tag{161}$$

$$\frac{\Delta, \sigma, E \circ [\mathsf{set}\ []] \vdash exp \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash \mathsf{set}\ exp \Downarrow \Delta', \sigma', ans} \tag{162}$$

$$\frac{\Delta, \sigma, E \circ [\mathsf{roll}_i^{con}\ []] \vdash exp \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash \mathsf{roll}_i^{con}\ exp \Downarrow \Delta', \sigma', ans} \tag{163}$$

$$\frac{\Delta, \sigma, E \circ [\mathsf{unroll}_i^{con}\ []] \vdash exp \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash \mathsf{unroll}_i^{con}\ exp \Downarrow \Delta', \sigma', ans} \tag{164}$$

$$\frac{\Delta, \sigma, E \circ [\mathsf{inj}_i^{con}\ []] \vdash exp \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash \mathsf{inj}_i^{con}\ exp \Downarrow \Delta', \sigma', ans} \tag{165}$$

$$\frac{\Delta, \sigma, E \circ [\mathsf{proj}_i\ []] \vdash exp \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash \mathsf{proj}_i\ exp \Downarrow \Delta', \sigma', ans} \tag{166}$$

$$\frac{\Delta, \sigma, E \circ [\mathsf{inj}_{[]}^{name}] \vdash exp \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash \mathsf{tag}^{name}\ exp \Downarrow \Delta', \sigma', ans} \tag{167}$$

$$\frac{\Delta, \sigma, E \circ [\mathsf{untag}^{name}\ []] \vdash exp \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash \mathsf{untag}^{name}\ exp \Downarrow \Delta', \sigma', ans} \tag{168}$$

$$\frac{\Delta, \sigma, E \circ [\mathsf{case}^{con}\ (exp_1, \cdots, exp_n)\ \mathsf{of}\ []\ \mathsf{end}] \vdash exp \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash \mathsf{case}^{con}\ (exp_1, \cdots, exp_n)\ \mathsf{of}\ exp\ \mathsf{end} \Downarrow \Delta', \sigma', ans} \tag{169}$$

$$\frac{\Delta, \sigma, E \circ [[].lbl] \vdash mod \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash mod.lbl \Downarrow \Delta', \sigma', ans} \tag{170}$$

$$\boxed{\Delta, \sigma, E \vdash mod \Downarrow \Delta', \sigma', ans}$$

$$\frac{\Delta, \sigma, E \circ [[lbl \triangleright var=[], sbnds]] \vdash exp \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash [lbl \triangleright var=exp, sbnds] \Downarrow \Delta', \sigma', ans} \tag{171}$$

$$\frac{con \vdash \Delta, \sigma, E \circ [[lbl \triangleright var=[], sbnds]] \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash [lbl \triangleright var=con, sbnds] \Downarrow \Delta', \sigma', ans} \tag{172}$$

$$\frac{\Delta, \sigma, E \circ [[lbl \triangleright var=[], sbnds]] \vdash mod \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash [lbl \triangleright var=mod, sbnds] \Downarrow \Delta', \sigma', ans} \tag{173}$$

$$\frac{\Delta, \sigma, E \circ [[]\ mod'] \vdash mod \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash mod\ mod' \Downarrow \Delta', \sigma', ans} \tag{174}$$

$$\frac{\Delta, \sigma, E \circ [[].lbl] \vdash mod \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash mod.lbl \Downarrow \Delta', \sigma', ans} \tag{175}$$

$$\frac{\Delta, \sigma, E \circ [[]{:}sig] \vdash mod \Downarrow \Delta', \sigma', ans}{\Delta, \sigma, E \vdash mod{:}lbl \Downarrow \Delta', \sigma', ans} \tag{176}$$

$$\boxed{val \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}$$

$$\frac{}{val \vdash \Delta, \sigma, [] \Downarrow \Delta, \sigma, \mathsf{VALUE}(val)} \tag{177}$$

$$\frac{\Delta, \sigma, E \circ [exp_{\mathrm{v}}\ []] \vdash exp \Downarrow \Delta', \sigma', ans}{exp_{\mathrm{v}} \vdash \Delta, \sigma, E \circ [[]\ exp] \Downarrow \Delta', \sigma', ans} \tag{178}$$

$$\frac{\begin{array}{c} exp_{\mathrm{v}}{}^k = \mathsf{fix}\ (lbl_i{=}(var'_i{:}var_i){\mapsto}exp_i)_{i=1}^n\ \mathsf{in}\ var_k\ \mathsf{end} \qquad (\forall k \in 1..n) \\ 1 \le i \le n \\ \Delta, \sigma, E \vdash [exp_{\mathrm{v}}{}^1/var'_1] \cdots [exp_{\mathrm{v}}{}^n/var'_n][exp_{\mathrm{v}}/var_i]exp_i \Downarrow \Delta', \sigma', ans \end{array}}{exp_{\mathrm{v}} \vdash \Delta, \sigma, E \circ [exp_{\mathrm{v}}{}^i\ []] \Downarrow \Delta', \sigma', ans} \tag{179}$$

$$\frac{\Delta, \sigma, E \circ [\{rbnds_{\mathrm{v}}, lbl \triangleright exp_{\mathrm{v}}, lbl' \triangleright [], rbnds\}] \vdash exp' \Downarrow \Delta', \sigma', ans}{exp_{\mathrm{v}} \vdash \Delta, \sigma, E \circ [\{rbnds_{\mathrm{v}}, lbl \triangleright [], lbl' \triangleright exp', rbnds\}] \Downarrow \Delta', \sigma', ans} \tag{180}$$

$$\frac{\{rbnds_{\mathrm{v}}, lbl \triangleright exp_{\mathrm{v}}\} \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{exp_{\mathrm{v}} \vdash \Delta, \sigma, E \circ [\{rbnds_{\mathrm{v}}, lbl \triangleright []\}] \Downarrow \Delta', \sigma', ans} \tag{181}$$

$$\frac{exp_{\mathrm{v}} \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{\{rbnds_{\mathrm{v}}, lbl \triangleright exp_{\mathrm{v}}, rbnds_{\mathrm{v}}{}'\} \vdash \Delta, \sigma, E \circ [[]\#lbl] \Downarrow \Delta', \sigma', ans} \tag{182}$$

$$\frac{exp_{\mathrm{v}} \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{exp_{\mathrm{v}} \vdash \Delta, \sigma, E \circ [\mathsf{handle}\,[]\,\mathsf{with}\,exp] \Downarrow \Delta', \sigma', ans} \tag{183}$$

$$\frac{\mathsf{raise}\,exp_{\mathrm{v}} \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{exp_{\mathrm{v}} \vdash \Delta, \sigma, E \circ [\mathsf{raise}\,[]] \Downarrow \Delta', \sigma', ans} \tag{184}$$

$$\frac{exp_{\mathrm{v}} \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{exp_{\mathrm{v}} \vdash \Delta, \sigma, E \circ [\mathsf{catch}\,[]\,\mathsf{with}\,exp] \Downarrow \Delta', \sigma', ans} \tag{185}$$

$$\frac{loc \notin \mathrm{dom}(\Delta) \qquad loc \vdash \Delta[loc{:}con], \sigma[loc \mapsto exp_{\mathrm{v}}], E \Downarrow \Delta', \sigma', ans}{exp_{\mathrm{v}} \vdash \Delta, \sigma, E \circ [\mathsf{ref}^{con}\,[]] \Downarrow \Delta', \sigma', ans} \tag{186}$$

$$\frac{exp_{\mathrm{v}} = \sigma(loc) \qquad exp_{\mathrm{v}} \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{exp_{\mathrm{v}} \vdash \Delta, \sigma, E \circ [\mathsf{get}\,con[]] \Downarrow \Delta', \sigma', ans} \tag{187}$$

$$\frac{\{\} \vdash \Delta, \sigma[loc \mapsto exp_{\mathrm{v}}], E \Downarrow \Delta', \sigma', ans}{\{1 \triangleright loc, 2 \triangleright exp_{\mathrm{v}}\} \vdash \Delta, \sigma, E \circ [\mathsf{set}\,con[]] \Downarrow \Delta', \sigma', ans} \tag{188}$$

$$\frac{exp_{\mathrm{v}} \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{exp_{\mathrm{v}} \vdash \Delta, \sigma, E \circ [\mathsf{roll}_i^{con}\,[]] \Downarrow \Delta', \sigma', ans} \tag{189}$$

$$\frac{exp_{\mathrm{v}} \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{exp_{\mathrm{v}} \vdash \Delta, \sigma, E \circ [\mathsf{unroll}_i^{con}\,[]] \Downarrow \Delta', \sigma', ans} \tag{190}$$

$$\frac{\mathsf{inj}_i^{con}\,exp_{\mathrm{v}} \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{exp_{\mathrm{v}} \vdash \Delta, \sigma, E \circ [\mathsf{inj}_i^{con}\,[]] \Downarrow \Delta', \sigma', ans} \tag{191}$$

$$\frac{exp_{\mathrm{v}} \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{\mathsf{inj}_i^{con}\,exp_{\mathrm{v}} \vdash \Delta, \sigma, E \circ [\mathsf{proj}_{con'}\,i[]] \Downarrow \Delta', \sigma', ans} \tag{192}$$

$$\frac{i \neq j \qquad \mathsf{fail} \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{\mathsf{inj}_i^{con}\,exp_{\mathrm{v}} \vdash \Delta, \sigma, E \circ [\mathsf{proj}_{con'}\,j[]] \Downarrow \Delta', \sigma', ans} \tag{193}$$

$$\frac{\mathsf{tag}^{name}\,exp_{\mathrm{v}} \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{exp_{\mathrm{v}} \vdash \Delta, \sigma, E \circ [\mathsf{tag}^{name}\,[]] \Downarrow \Delta', \sigma', ans} \tag{194}$$

$$\frac{exp_{\mathrm{v}} \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{\mathsf{tag}^{name}\ exp_{\mathrm{v}} \vdash \Delta, \sigma, E \circ [\mathsf{untag}^{name}\ []] \Downarrow \Delta', \sigma', ans} \tag{195}$$

$$\frac{name \neq name' \qquad \mathsf{fail} \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{\mathsf{tag}^{name}\ exp_{\mathrm{v}} \vdash \Delta, \sigma, E \circ [\mathsf{untag}^{name'}\ []] \Downarrow \Delta', \sigma', ans} \tag{196}$$

$$\frac{\Delta, \sigma, E \vdash\ exp_i\ exp_{\mathrm{v}} \Downarrow \Delta', \sigma', ans}{\mathsf{inj}_i^{con}\ exp_{\mathrm{v}} \vdash \Delta, \sigma, E \circ [\mathsf{case}^{con'}\ (exp_1, \cdots, exp_n)\ \mathsf{of}\ []\ \mathsf{end}] \Downarrow \Delta', \sigma', ans} \tag{197}$$

$$\frac{\Delta, \sigma, E \vdash [mod_{\mathrm{v}}/var]exp \Downarrow \Delta', \sigma', ans}{mod_{\mathrm{v}} \vdash \Delta, \sigma, E \circ [\mathsf{let}\ var{=}[]\ \mathsf{in}\ exp\ \mathsf{end}] \Downarrow \Delta', \sigma', ans} \tag{198}$$

$$\frac{\Delta, \sigma, E \circ [mod_{\mathrm{v}}\ []] \vdash mod \Downarrow \Delta', \sigma', ans}{mod_{\mathrm{v}} \vdash \Delta, \sigma, E \circ [[]\ mod'] \Downarrow \Delta', \sigma', ans} \tag{199}$$

$$\frac{\Delta, \sigma, E \vdash [mod_{\mathrm{v}}/var]mod \Downarrow \Delta', \sigma', ans}{mod_{\mathrm{v}} \vdash \Delta, \sigma, E \circ [(\lambda var{:}sig.mod)\ []] \Downarrow \Delta', \sigma', ans} \tag{200}$$

$$\frac{val \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{[sbnds_{\mathrm{v}}, lbl{\triangleright}val, sbnds_{\mathrm{v}}'] \vdash \Delta, \sigma, E \circ [[].lbl] \Downarrow \Delta', \sigma', ans} \tag{201}$$

$$\frac{mod_{\mathrm{v}} \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{mod_{\mathrm{v}} \vdash \Delta, \sigma, E \circ [[]{:}sig] \Downarrow \Delta', \sigma', ans} \tag{202}$$

Rule 202: Forgetting of type abbreviations has no run-time effect.

$$\frac{\Delta, \sigma, E \circ [[sbnds_{\mathrm{v}}, lbl{\triangleright}var{=}val, lbl'{\triangleright}var'{=}[], [val/var]sbnds]] \vdash [val/var]exp \Downarrow \Delta', \sigma', ans}{val \vdash \Delta, \sigma, E \circ [[sbnds_{\mathrm{v}}, lbl{\triangleright}var{=}[], lbl'{\triangleright}var'{=}\exp, [val/var]sbnds]] \Downarrow \Delta', \sigma', ans} \tag{203}$$

$$\frac{\Delta, \sigma, E \circ [[sbnds_{\mathrm{v}}, lbl{\triangleright}var{=}val, lbl'{\triangleright}var'{=}[], [val/var]sbnds]] \vdash [val/var]mod \Downarrow \Delta', \sigma', ans}{val \vdash \Delta, \sigma, E \circ [[sbnds_{\mathrm{v}}, lbl{\triangleright}var{=}[], lbl'{\triangleright}var'{=}mod, [val/var]sbnds]] \Downarrow \Delta', \sigma', ans} \tag{204}$$

$$\frac{con \vdash \Delta, \sigma, E \circ [[sbnds_{\mathrm{v}}, lbl{\triangleright}var{=}val, lbl'{\triangleright}var'{=}[], [val/var]sbnds]] \Downarrow \Delta', \sigma', ans}{val \vdash \Delta, \sigma, E \circ [[sbnds_{\mathrm{v}}, lbl{\triangleright}var{=}[], lbl'{\triangleright}var'{=}con, [val/var]sbnds]] \Downarrow \Delta', \sigma', ans} \tag{205}$$

$$\frac{[sbnds_{\mathrm{v}}, lbl{\triangleright}var{=}val] \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{val \vdash \Delta, \sigma, E \circ [[sbnds_{\mathrm{v}}, lbl{\triangleright}var{=}[]]] \Downarrow \Delta', \sigma', ans} \tag{206}$$

$$\boxed{\mathsf{raise}\ exp_{\mathrm{v}} \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}$$

$$\frac{}{\mathsf{raise}\ exp_{\mathrm{v}} \vdash \Delta, \sigma, [] \Downarrow \Delta, \sigma, \mathsf{UNCAUGHT}(exp_{\mathrm{v}})} \tag{207}$$

$$\frac{\mathsf{raise}\ exp_{\mathrm{v}} \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{\mathsf{raise}\ exp_{\mathrm{v}} \vdash \Delta, \sigma, E \circ R \Downarrow \Delta', \sigma', ans} \tag{208}$$

$$\frac{\Delta, \sigma, E \vdash exp\ exp_{\mathrm{v}} \Downarrow \Delta', \sigma', ans}{\mathsf{raise}\ exp_{\mathrm{v}} \vdash \Delta, \sigma, E \circ [\mathsf{handle}\,[]\,\mathsf{with}\ exp] \Downarrow \Delta', \sigma', ans} \tag{209}$$

$$\frac{\mathsf{raise}\ exp_{\mathrm{v}} \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{\mathsf{raise}\ exp_{\mathrm{v}} \vdash \Delta, \sigma, E \circ [\mathsf{catch}\,[]\,\mathsf{with}\ exp] \Downarrow \Delta', \sigma', ans} \tag{210}$$

$$\boxed{\mathsf{fail} \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}$$

$$\frac{}{\mathsf{fail} \vdash \Delta, \sigma, [] \Downarrow \Delta, \sigma, \mathsf{FAIL}} \tag{211}$$

$$\frac{\mathsf{fail} \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{\mathsf{fail} \vdash \Delta, \sigma, E \circ R \Downarrow \Delta', \sigma', ans} \tag{212}$$

$$\frac{\mathsf{fail} \vdash \Delta, \sigma, E \Downarrow \Delta', \sigma', ans}{\mathsf{fail} \vdash \Delta, \sigma, E \circ [\mathsf{handle}\,[]\,\mathsf{with}\ exp] \Downarrow \Delta', \sigma', ans} \tag{213}$$

$$\frac{\Delta, \sigma, E \vdash exp \Downarrow \Delta', \sigma', ans}{\mathsf{fail} \vdash \Delta, \sigma, E \circ [\mathsf{catch}\,[]\,\mathsf{with}\ exp] \Downarrow \Delta', \sigma', ans} \tag{214}$$

$$\boxed{\Delta, \sigma \vdash tbnds \Downarrow \Delta', \sigma', Ans}$$

$$\frac{}{\Delta, \sigma \vdash \cdot \Downarrow \Delta, \sigma, \mathsf{VALUES}(\cdot)} \tag{215}$$

$$\frac{\begin{array}{c}\Delta, \sigma, [] \vdash mod \Downarrow \Delta', \sigma', \mathsf{VALUE}(mod_{\mathrm{v}}) \\ \Delta', \sigma' \vdash [mod_{\mathrm{v}}/var]tbnds \Downarrow \Delta'', \sigma'', \mathsf{VALUES}(tbnds')\end{array}}{\Delta, \sigma \vdash lbl \triangleright var{=}mod, tbnds \Downarrow \Delta'', \sigma'', \mathsf{VALUES}(lbl \triangleright var{=}mod_{\mathrm{v}}, tbnds')} \tag{216}$$

$$\frac{\begin{array}{c}\Delta, \sigma, [] \vdash mod \Downarrow \Delta', \sigma', \mathsf{VALUE}(mod_{\mathrm{v}}) \\ \Delta', \sigma' \vdash [mod_{\mathrm{v}}/var]tbnds \Downarrow \Delta'', \sigma'', \mathsf{UNCAUGHT}(exp_{\mathrm{v}})\end{array}}{\Delta, \sigma \vdash lbl \triangleright var{=}mod, tbnds \Downarrow \Delta'', \sigma'', \mathsf{UNCAUGHT}(exp_{\mathrm{v}})} \tag{217}$$

$$\frac{\Delta, \sigma, [] \vdash mod \Downarrow \Delta', \sigma', \mathsf{VALUE}(mod_{\mathrm{v}}) \quad \Delta', \sigma' \vdash [mod_{\mathrm{v}}/var]tbnds \Downarrow \Delta'', \sigma'', \mathsf{FAIL}}{\Delta, \sigma \vdash lbl \triangleright var = mod, tbnds \Downarrow \Delta'', \sigma'', \mathsf{FAIL}} \tag{218}$$

$$\frac{\Delta, \sigma, [] \vdash mod \Downarrow \Delta', \sigma', \mathsf{UNCAUGHT}(exp_{\mathrm{v}})}{\Delta, \sigma \vdash lbl \triangleright var = mod, tbnds \Downarrow \Delta'', \sigma'', \mathsf{UNCAUGHT}(exp_{\mathrm{v}})} \tag{219}$$

$$\frac{\Delta, \sigma, [] \vdash mod \Downarrow \Delta', \sigma', \mathsf{FAIL}}{\Delta, \sigma \vdash lbl \triangleright var = mod, tbnds \Downarrow \Delta'', \sigma'', \mathsf{FAIL}} \tag{220}$$

$$\frac{\Delta, \sigma \vdash lbl \triangleright var = mod, tbnds \Downarrow \Delta'', \sigma'', \mathsf{VALUES}(tbnds')}{\Delta, \sigma \vdash tbnds \Downarrow \Delta'', \sigma'', \mathsf{VALUES}(lbl \triangleright var = sig, tbnds')} \tag{221}$$

7 External Language

7.1 Notation

As in the *Definition*, optional elements elements are enclosed by single brackets $\langle \cdots \rangle$ or double brackets $\langle\langle \cdots \rangle\rangle$. For the purposes of this grammar, all optional choices are completely independent.

7.2 Grammar of the Abstract Syntax

$$
\begin{array}{rcl}
\mathit{expr} & ::= & \mathit{scon} \\
& | & \mathit{longid} \\
& | & \{\mathit{lab}_1 \texttt{ = } \mathit{expr}_1 \texttt{,} \cdots \texttt{,} \mathit{lab}_n \texttt{ = } \mathit{expr}_n\texttt{}\} \\
& | & \texttt{let } \mathit{strdec} \texttt{ in } \mathit{expr} \texttt{ end} \\
& | & \mathit{expr}\ \mathit{expr}' \\
& | & \mathit{expr} \texttt{ : } \mathit{ty} \\
& | & \mathit{expr} \texttt{ handle } \mathit{match} \\
& | & \texttt{raise } \mathit{expr} \\
& | & \texttt{fn } \mathit{match} \\
\\
\mathit{mrule} & ::= & \mathit{pat} \texttt{ => } \mathit{expr} \\
\\
\mathit{match} & ::= & \mathit{mrule} \\
& | & \mathit{mrule} \texttt{ | } \mathit{match} \\
\\
\mathit{strdec} & ::= & \texttt{val (}\mathit{tyvar}_1 \texttt{,} \cdots \texttt{,} \mathit{tyvar}_n\texttt{) } \mathit{pat} \texttt{ = } \mathit{exp} \\
& | & \texttt{val (}\mathit{tyvar}_1 \texttt{,} \cdots \texttt{,} \mathit{tyvar}_n\texttt{) rec } \mathit{pat} \texttt{ = } \mathit{exp} \\
& | & \mathit{strdec}_1\ \mathit{strdec}_2 \\
& | & \texttt{open } \mathit{longid}_1 \cdots \mathit{longid}_n \\
& | & \texttt{exception } \mathit{id} \\
& | & \texttt{exception } \mathit{id} \texttt{ of } \mathit{ty} \\
& | & \texttt{exception } \mathit{id} \texttt{ = } \mathit{longid} \\
& | & \texttt{local } \mathit{strdec}_1 \texttt{ in } \mathit{strdec}_2 \texttt{ end} \\
& | & \texttt{type } \mathit{tybind} \\
& | & \texttt{datatype } \mathit{datbind} \\
& | & \texttt{structure } \mathit{strbind} \\
& | & \texttt{functor } \mathit{funbind}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{tybind} & ::= & \langle (\mathit{tyvar}_1, \cdots, \mathit{tyvar}_n) \rangle\ \mathit{tycon}\ \texttt{=}\ \mathit{ty}\ \langle \langle \texttt{and}\ \mathit{tybind} \rangle \rangle \\
\mathit{datbind} & ::= & (\mathit{tyvar}_1, \cdots, \mathit{tyvar}_n)\ \mathit{tycon}\ \texttt{=}\ \mathit{conbind}\ \langle \texttt{and}\ \mathit{datbind} \rangle \\
\mathit{conbind} & ::= & \mathit{id}\ \langle \texttt{of}\ \mathit{ty} \rangle\ \langle \langle \texttt{|}\ \mathit{conbind} \rangle \rangle \\
\\
\mathit{strexp} & ::= & \texttt{struct}\ \mathit{strdec}\ \texttt{end} \\
& | & \mathit{funid}\ \texttt{(}\mathit{strexp}\texttt{)} \\
& | & \mathit{strexp}\ \texttt{:}\ \mathit{sigexp} \\
& | & \mathit{strexp}\ \texttt{:>}\ \mathit{sigexp} \\
& | & \texttt{let}\ \mathit{strdec}\ \texttt{in}\ \mathit{strexp}\ \texttt{end} \\
\\
\mathit{spec} & ::= & \texttt{val}\ \mathit{id}\ \texttt{:}\ \mathit{ty} \\
& | & \texttt{type}\ \mathit{typdesc} \\
& | & \texttt{datatype}\ \mathit{datbind} \\
& | & \texttt{exception}\ \mathit{id} \\
& | & \texttt{exception}\ \mathit{id}\ \texttt{of}\ \mathit{ty} \\
& | & \texttt{structure}\ \mathit{strid}\ \texttt{:}\ \mathit{sigexp} \\
& | & \texttt{functor}\ \mathit{funid}\ \texttt{(}\mathit{strid}\ \texttt{:}\ \mathit{sigexp}\texttt{)}\ \texttt{:}\ \mathit{sigexp}' \\
& | & \texttt{include}\ \mathit{sigexp} \\
& | & \mathit{spec}_1\ \mathit{spec}_2 \\
& | & \mathit{spec}\ \texttt{sharing type}\ \mathit{longid}_1\ \texttt{=}\ \mathit{longid}_2 \\
\\
\mathit{typdesc} & ::= & \langle (\mathit{tyvar}_1, \cdots, \mathit{tyvar}_n) \rangle\ \mathit{tycon}\ \langle \langle \texttt{and}\ \mathit{typdesc} \rangle \rangle \\
& | & \langle (\mathit{tyvar}_1, \cdots, \mathit{tyvar}_n) \rangle\ \mathit{tycon}\ \texttt{=}\ \mathit{ty}\ \langle \langle \texttt{and}\ \mathit{typdesc} \rangle \rangle \\
\\
\mathit{sigexp} & ::= & \texttt{sig}\ \mathit{spec}\ \texttt{end} \\
& | & \mathit{sigid} \\
& | & \mathit{sigexp}\ \texttt{where type}\ \langle (\mathit{tyvar}_1, \cdots, \mathit{tyvar}_n) \rangle\ \mathit{longtycon}\ \texttt{=}\ \mathit{ty} \\
& | & \mathit{sigexp}\ \texttt{where type}\ \langle (\mathit{tyvar}_1, \cdots, \mathit{tyvar}_n) \rangle\ \mathit{longtycon}\ \texttt{=}\ \texttt{own}\ \mathit{ty} \\
\\
\mathit{pat} & ::= & \mathit{scon} \\
& | & \mathit{longid} \\
& | & \texttt{\_} \\
& | & \mathit{pat}\ \texttt{:}\ \mathit{ty} \\
& | & \mathit{longid}\ \mathit{pat} \\
& | & \texttt{\{}\mathit{lab}_1\ \texttt{=}\ \mathit{pat}_1, \cdots, \mathit{lab}_n\ \texttt{=}\ \mathit{pat}_n \langle \texttt{,...} \rangle \texttt{\}} \\
& | & \mathit{pat}_1\ \texttt{as}\ \mathit{pat}_2 \\
& | & \texttt{ref}\ \mathit{pat}
\end{array}
$$

$$
\begin{array}{rcl}
ty & ::= & tyvar \\
& | & \texttt{\{}lab_1 \texttt{ : } expr_1 \texttt{,} \cdots \texttt{,} lab_n \texttt{ : } expr_n \texttt{\}} \\
& | & \langle \texttt{(}ty_1 \texttt{,} \cdots \texttt{,} ty_n \texttt{)} \rangle \; longtycon \\
& | & ty \texttt{ -> } ty'
\end{array}
$$

$$
\begin{array}{rcl}
sigbind & ::= & sigid \texttt{ = } sigexp \; \langle \texttt{and } sigbind \rangle \\
strbind & ::= & strid \texttt{ = } strexp \; \langle \texttt{and } strbind \rangle \\
funbind & ::= & funid \texttt{ (}strid \texttt{ : } sigexp\texttt{) = } strexp \; \langle \texttt{and } funbind \rangle
\end{array}
$$

$$
\begin{array}{rcl}
topdec & ::= & \texttt{signature } sigbind \\
& | & \texttt{structure } strbind \\
& | & \texttt{functor } funbind \\
& | & topdec_1 \; topdec_2
\end{array}
$$

7.3 Syntactic Restrictions

- No pattern may contain the same *id* twice. No record expression, pattern, or type may contain the same *lab* twice. No *tyvar* may appear more than once in a single sequence.

- Any type variable occuring in a *conbind* must also appear in the enclosing *datbind*. Any type variable appearing in the *ty* of a `where type` must appear in the type variable sequence.

- No `val`, `type`, `datatype`, `exception`, `structure`, `signature`, or `functor` binding or specification may bind the same identifier twice; this applies also to value constructors within a *datbind*.

- In a `val rec` declaration, the pattern must be of the form

  $$\{lab_1\texttt{=}id_1\texttt{,}\cdots\texttt{,}lab_n\texttt{=}id_n\texttt{}\}$$

  and the expression must be of the form

  $$\{lab_1\texttt{=fn }match_1\texttt{,}\cdots\texttt{,}lab_n\texttt{=fn }match_n\texttt{}\}.$$

  Hence the "..." EL-notation may not appear in the pattern.

- We disallow "polymorphic recursion" in a *datbind*; any recursive use of a datatype must be applied to exactly the same type variables that scope the definition of that datatype. This disallows such code as

```
datatype 'a invalid = A | B of ('a * 'a) invalid
```

8 The Translation

8.1 Introduction

In addition to type-checking and type-reconstruction, the elaborator performs the following tasks:

1. Datatypes are expanded into structures and signatures whose components include

   - an abstract implementation type (which is chosen to be a recursive sum type);
   - operations corresponding to constructors as values;
   - operations corresponding to constructors as patterns (deconstructors);
   - a "case" function that does branching but no destructuring.

   In particular, a datatype translates to a type and a series of constructors. The constructors are little structures containing two components; the "mk" component is the injection into the datatype while the "km" component is the projection from the datatype. Note that the mk component is always total, but the km component may fail if there is more than one constructor. (Exception constructors are handled similarly to datatype constructors, except that their "mk" and "km" components are monomorphic.)

   The "generativity" of datatypes is handled via signature ascription; the type is made opaque in the corresponding signature, and is therefore not equivalent to any other type. The matching of datatypes in signatures reduces to the matching of substructures.

2. Polymorphism is encoded as a use of the modules system. Polymorphic values are translated into functors, which are explicitly instantiated with structures of types when necessary. More precisely, the functor takes a structure containing type constructors of kind Ω, and returns a structure whose single component (with label "it") is the polymorphic value made monomorphic by instantiating it with the given types.

3. Patterns are expanded into uses of the appropriate record projections and datatype deconstructors. Thus the translation specifies a reference pattern compiler.

4. Each series of external language bindings (*strdec*) translates into a structure, containing a component for every variable bound in the external language. External language *identifiers* correspond to internal language *labels*.

5. Some structure labels are explicitly marked with an asterisk (lbl^*). This indicates that the structure is "open" for the purposes of identifier lookup. (See the lookup rules for more details.)

6. In structures, `open` declarations are handled by translating field names into paths. Thus there is no run-time copying/flattening of structures due to `open`.

7. All coercive aspects of the signature matching relation (the reordering and forgetting of components) are handled by introducing explicit coercion functors witnessing the relation. This makes the order and number of components a structure apparent from its signature.

8.2 Notation

- The overbar function $\overline{\cdot}$ maps each EL identifier to an IL label. We assume that this function is injective, that the range is coinfinite in the set of IL labels, and that identifiers of different classes map to different labels. In particular, we assume that the parser distinguishes between the classes of expression variables, type constructors, type variables, structure identifiers, signature identifiers, and functor identifiers. However, we do not distinguish between an identifier being used as an expression variable, datatype constructor, or exception constructor.

  The special labels "mk", "km", "case", and "it" used by translation are not in the range of the overbar mapping, and labels chosen to be "fresh" are similarly not in the range of the mapping.

  We extend the overbar mapping component-wise to long identifiers, which thus map to sequences of labels.

- Optional elements are enclosed in single or double angle brackets. For each rule, either all or none of the elements in single angle brackets must be present, and similarly all or none of the elements in double angle brackets must be present. Single and double angle brackets in the same rule represent two *independent* choices.

- In some cases, the optional element notation is insufficient. Therefore, we have the additional notation

$$\left\{\begin{array}{c} element_1 \\ \text{or} \\ element_2 \end{array}\right\}$$

  which means that either $element_1$ or $element_2$ must be present. If there are multiple such choices in a single rule, this means that either the first element should always be chosen in all cases, or the second element must be chosen in all cases.

  An extension of this notation gives the choices subscripts. Then all choices with the same subscript must agree (all first element or all second element) but two choices with different subscripts are completely independent.

- The translation maintains a translation context Γ. When Γ appears in an IL judgment where *decs* is expected, there is an implicit coercion which drops all top-level labels and all signature declarations.

8.3 Initial Basis

The translation assumes the presence of a structure $basis{:}sig_{basis}$ serving as the initial basis for the internal language. We use the components:

$basis.\overline{\text{bind}}$	Bind exception
$basis.\overline{\text{match}}$	Match exception
$basis.\overline{\text{bool}}$	Bool datatype, with constructors $basis.\overline{\text{true}}$ and $basis.\overline{\text{false}}$
$basis.\text{equal}_{con}$	Equality on base types *con* for pattern-matching

8.4 Judgment Forms

Judgment...	*Meaning...*
$\Gamma \vdash expr \rightsquigarrow exp : con$	Translation of an expression
$\Gamma \vdash match \rightsquigarrow exp : con$	Translation of a pattern match
$\Gamma \vdash strdec \rightsquigarrow mod : sig$	Translation of a declaration
$\Gamma \vdash strexp \rightsquigarrow mod : sig$	Translation of a structure expression
$\Gamma \vdash spec \rightsquigarrow [sdecs] : \mathsf{Sig}$	Translation of a signature specification
$\Gamma \vdash sigexp \rightsquigarrow sig : \mathsf{Sig}$	Translation of a signature expression
$\Gamma \vdash topdec \rightsquigarrow tbnds : tdecs$	Translation of a program
$\Gamma \vdash ty \rightsquigarrow con : \Omega$	Translation of a type
$\Gamma \vdash tybind \rightsquigarrow mod : sig$	Translation of a `type` definition
$\Gamma \vdash datbind \rightsquigarrow mod : sig$	Translation of a `datatype` definition
$\Gamma \vdash pat \Leftarrow exp : con \mid exp' \rightsquigarrow mod : sig$	Pattern Compilation
$\Gamma \vdash_{\text{ctx}} lbl \rightsquigarrow path : \Theta / sig$	Lookup in a translation context
$\Gamma, path{:}sdecs \vdash_{\text{sig}} lbl \rightsquigarrow lbls : \Theta$	Lookup in a signature
$decs \vdash_{\text{sub}} sig_0 \preceq sig \rightsquigarrow (\lambda var_0{:}sig_0.mod) : ((var_0{:}sig_0) \rightarrow sig')$	Witness to EL-signature subtyping
$sig \vdash_{\text{wt}} lbls := con : knd \rightsquigarrow sig' : \mathsf{Sig}$	Signature Patching (`where type`)
$sig \vdash_{\text{sh}} lbls := lbls' : knd \rightsquigarrow sig' : \mathsf{Sig}$	Signature Patching (`sharing type`)

8.5 Inference Rules

Expressions

$\boxed{\Gamma \vdash expr \rightsquigarrow exp : con}$

$$\frac{}{\Gamma \vdash scon \rightsquigarrow scon : \text{type}(scon)} \tag{222}$$

$$\frac{\Gamma \vdash_{\text{ctx}} \overline{longid} \rightsquigarrow path : con}{\Gamma \vdash longid \rightsquigarrow path : con} \tag{223}$$

Rule 223: This rule handles monomorphic identifiers.

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{ctx}} \overline{longid} \rightsquigarrow path : sig \\ \Gamma \vdash sig \leq (sig_{poly} \rightarrow [\mathrm{it} \triangleright con]) : \mathsf{Sig} \\ \Gamma \vdash mod_{poly} \downarrow sig_{poly}\end{array}}{\Gamma \vdash longid \rightsquigarrow path(mod_{poly}).\mathrm{it} : con} \tag{224}$$

Rule 224:

- This rule handles instantiation of polymorphic identifiers.
- Recall that all polymorphic functions are translated into total functors whose body contains a single component with the label "it".
- The module mod_{poly} in Rule 224 is the structure of types that we "guess" to instantiate the polymorphic function.

$$\frac{\Gamma \vdash_{\mathrm{ctx}} \overline{longid} \rightsquigarrow path : sig \qquad \Gamma \vdash path.\mathrm{mk} : con}{\Gamma \vdash longid \rightsquigarrow path : con} \tag{225}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{ctx}} \overline{longid} \rightsquigarrow path : sig \qquad \Gamma \vdash path.\mathrm{mk} : sig_{\mathrm{mk}} \\ \Gamma \vdash sig_{\mathrm{mk}} \leq (sig_{poly} \rightarrow [\mathrm{it} \triangleright con]) : \mathsf{Sig} \\ \Gamma \vdash mod_{poly} \downarrow sig_{poly}\end{array}}{\Gamma \vdash longid \rightsquigarrow ((path.\mathrm{mk})mod_{poly}).\mathrm{it} : con} \tag{226}$$

Rules 225 and 226: Recall that constructors translate to structures with mk and km components. When a constructor is used as a value, we translate it to the mk operation, making the type partial in the process.

$$\frac{\Gamma \vdash expr_1 \rightsquigarrow exp_1 : con_1 \quad \cdots \quad \Gamma \vdash expr_n \rightsquigarrow exp_n : con_n}{\begin{array}{l}\Gamma \vdash \{lab_1 \texttt{ = } expr_1, \cdots, lab_n \texttt{ = } expr_n\} \rightsquigarrow \\ \quad \{\overline{lab_1} \triangleright exp_1, \cdots, \overline{lab_n} \triangleright exp_n\} : \{\overline{lab_1} \triangleright con_1, \cdots, \overline{lab_n} \triangleright con_n\}\end{array}} \tag{227}$$

$$\frac{\begin{array}{c}\Gamma \vdash dec \rightsquigarrow mod : sig \\ var \notin \mathrm{bound}(\Gamma) \qquad \Gamma, lbl^* \triangleright var{:}sig \vdash expr \rightsquigarrow exp : con' \\ \Gamma, var{:}sig \vdash con' \equiv con : \Omega \qquad \Gamma \vdash con : \Omega\end{array}}{\Gamma \vdash \texttt{let } dec \texttt{ in } expr \texttt{ end} \rightsquigarrow \mathsf{let}\ var{=}mod\ \mathsf{in}\ exp\ \mathsf{end} : con} \tag{228}$$

Rule 228:

- Declarations are uniformly translated to components of a structure; the "starred structure" convention is used here to make these components accessible by name.
- This rule prohibits the type of the body from depending on abstract types defined locally—in particular, datatypes cannot escape their scope.

$$\frac{\Gamma \vdash expr \rightsquigarrow exp : con'' \qquad \Gamma \vdash expr' \rightsquigarrow exp' : con' \qquad \Gamma \vdash exp : con' \rightharpoonup con}{\Gamma \vdash expr\ expr' \rightsquigarrow exp\ exp' : con} \tag{229}$$

Rule 229: We check that the application is well-typed in the IL. (This rule works whether *exp* is partial or total.)

$$\frac{\Gamma \vdash expr \rightsquigarrow exp : con \qquad \Gamma \vdash ty \rightsquigarrow con' : \Omega \qquad \Gamma \vdash con \equiv con' : \Omega}{\Gamma \vdash expr\ :\ ty \rightsquigarrow exp : con} \tag{230}$$

$$\frac{\Gamma \vdash expr \rightsquigarrow exp : con \qquad \Gamma \vdash match \rightsquigarrow exp' : \mathsf{Any} \rightharpoonup con \qquad var \notin \mathrm{bound}(\Gamma)}{\begin{array}{l}\Gamma \vdash expr\ \mathtt{handle}\ match \rightsquigarrow \\ \quad \mathsf{handle}\ exp\ \mathsf{with}\ \lambda\, var{:}\mathsf{Any}.(\mathsf{catch}\ exp'\ var\ \mathsf{with}\ \mathsf{raise}\ var) : con\end{array}} \tag{231}$$

Rule 231: The handling expression *exp′ var* may fail if the handler pattern does not match the exception raised, in which case we propagate the exception.

$$\frac{\Gamma \vdash expr \rightsquigarrow exp : \mathsf{Any}}{\Gamma \vdash \mathtt{raise}\ expr \rightsquigarrow \mathsf{raise}\ exp : con} \tag{232}$$

$$\frac{\Gamma \vdash match \rightsquigarrow exp : con_1 \rightharpoonup con_2 \qquad var \notin \mathrm{bound}(\Gamma)}{\begin{array}{l}\Gamma \vdash \mathtt{fn}\ match \rightsquigarrow \\ \quad \lambda\, var{:}con_1.(\mathsf{catch}\ exp\ var\ \mathsf{with}\ \mathsf{raise}\ basis.\overline{\mathrm{Match}}) : con_1 \rightharpoonup con_2\end{array}} \tag{233}$$

Rule 233: The expression *exp var* will fail if the match fails; here we turn failure into a $basis.\overline{\mathrm{Match}}$ exception.

Matches

$\boxed{\Gamma \vdash match \rightsquigarrow exp : con}$

$$\frac{\begin{array}{c} var, var' \notin \mathrm{bound}(\Gamma) \\ \Gamma \vdash con' : \Omega \\ \Gamma \vdash pat \Leftarrow var' : con' \mid \mathsf{fail} \rightsquigarrow mod : sig \\ \Gamma, lbl^* \triangleright var{:}sig \vdash expr \rightsquigarrow exp : con \end{array}}{\begin{array}{l} \Gamma \vdash pat \texttt{ => } expr \rightsquigarrow \\ \quad \lambda\, var'{:}con'.\mathsf{let}\ var{=}mod\ \mathsf{in}\ exp\ \mathsf{end} : con' \rightharpoonup con \end{array}} \tag{234}$$

Rule 234:

- The result of translating a match is a function that may fail if the match fails.
- We know that $\Gamma \vdash con' \rightharpoonup con : \Omega$ because the pattern-compilation rules create a structure with no type components.

$$\frac{\begin{array}{c} var \notin \mathrm{bound}(\Gamma) \\ \Gamma \vdash mrule \rightsquigarrow exp : con' \rightharpoonup con \\ \Gamma \vdash match \rightsquigarrow exp' : con' \rightharpoonup con \end{array}}{\begin{array}{l} \Gamma \vdash mrule \texttt{ | } match \rightsquigarrow \\ \quad \lambda\, var{:}con'.\mathsf{catch}\ exp\ var\ \mathsf{with}\ exp'\ var : con' \rightharpoonup con \end{array}} \tag{235}$$

Declarations

$\boxed{\Gamma \vdash strdec \rightsquigarrow mod : sig}$

$$\frac{\begin{array}{c} \Gamma \vdash expr \rightsquigarrow exp : con \\ \Gamma \vdash pat \Leftarrow exp : con \mid \mathsf{raise}\, basis.\overline{\mathrm{Bind}} \rightsquigarrow mod : sig \end{array}}{\Gamma \vdash \texttt{val ()}\ pat \texttt{ = } expr \rightsquigarrow mod : sig} \tag{236}$$

Rule 236: This is the monomorphic, non-recursive case.

$$\frac{\begin{array}{c} lbl^* \notin \mathrm{dom}(\Gamma) \qquad lbl' \text{ fresh} \\ sig_{poly} = [\overline{tyvar_1}\triangleright\Omega, \cdots, \overline{tyvar_n}\triangleright\Omega, lbl'_1\triangleright\Omega, \cdots, lbl'_m\triangleright\Omega] \\ \Gamma, lbl^*\triangleright var_{poly}{:}sig_{poly} \vdash expr \rightsquigarrow exp : con \\ \Gamma, lbl^*\triangleright var_{poly}{:}sig_{poly} \vdash pat \Leftarrow exp : con \mid \mathsf{raise}\, basis.\overline{\mathrm{Bind}} \rightsquigarrow mod : sig \\ \Gamma, lbl^*\triangleright var_{poly}{:}sig_{poly} \vdash mod \downarrow sig \\ sig = [lbl_1\triangleright con_1, \cdots, lbl_k\triangleright con_k] \end{array}}{\begin{array}{l} \Gamma \vdash \texttt{val}\ (tyvar_1, \cdots tyvar_n)\ pat\ \texttt{=}\ expr \rightsquigarrow \\ \quad [lbl'\triangleright var'{=}(\lambda var{:}sig_{poly}.mod), \\ \quad\ lbl_1\triangleright(\lambda var{:}sig_{poly}.[\mathrm{it}\triangleright(var'\,(var)).lbl'_1]), \\ \quad\ \vdots \\ \quad\ lbl_k\triangleright(\lambda var{:}sig_{poly}.[\mathrm{it}\triangleright(var'\,(var)).lbl'_k])] : \\ \quad [lbl'\triangleright var'{:}((var{:}sig_{poly})\rightarrow sig), \\ \quad\ lbl_1\triangleright((var{:}sig_{poly})\rightarrow[\mathrm{it}\triangleright con_1]), \\ \quad\ \vdots \\ \quad\ lbl_k\triangleright((var{:}sig_{poly})\rightarrow[\mathrm{it}\triangleright con_k])] \end{array}} \quad (237)$$

Rule 237:

- This rule handles polymorphic, non-recursive `val` bindings.
- We assume a prepass which annotates `val` declarations with the explicit type variables implicitly scoped by that declaration.

$$\frac{\begin{array}{c} var_{poly} \notin \mathrm{bound}(\Gamma) \\ sig_{poly} = [\overline{tyvar_1}\triangleright\Omega, \cdots, \overline{tyvar_n}\triangleright\Omega, lbl_1\triangleright\Omega, \cdots, lbl_m\triangleright\Omega] \\ \Gamma' = \Gamma, lbl^*\triangleright var_{poly}{:}sig_{poly}, \overline{id_1}\triangleright var'_1{:}con_1 \rightharpoonup con'_1, \cdots, \overline{id_n}\triangleright var'_n{:}con_n \rightharpoonup con'_n \\ \Gamma' \vdash match_1 \rightsquigarrow \lambda\, var_1{:}con_1.exp_1 : con_1 \rightharpoonup con'_1 \\ \vdots \\ \Gamma' \vdash match_m \rightsquigarrow \lambda\, var_n{:}con_n.exp_n : con_n \rightharpoonup con'_n \\ exp_k = \mathsf{fix}\ var'_1{=}(var_1{:}con_1){\mapsto}exp_1, \cdots, var'_m{=}(var_m{:}con_m){\mapsto}exp_m\ \mathsf{in}\ var'_k\ \mathsf{end} \end{array}}{\begin{array}{l} \Gamma \vdash \texttt{val}\ (tyvar_1, \cdots tyvar_n)\ \texttt{rec}\ \{(lab_i\texttt{=}id_i)_{i=1}^n\}\ \texttt{=}\ \{(lab_i\texttt{=}\texttt{fn}\ match_i)_{i=1}^n\} \rightsquigarrow \\ \quad [\overline{id_1}\triangleright(\lambda var_{poly}{:}sig_{poly}.[\mathrm{it}\triangleright exp_1]) \\ \quad \vdots \\ \quad \overline{id_m}\triangleright(\lambda var_{poly}{:}sig_{poly}.[\mathrm{it}\triangleright exp_m])] : \\ \quad [\overline{id_1}\triangleright((var_{poly}{:}sig_{poly}){\rightarrow}[\mathrm{it}\triangleright con_1 \rightharpoonup con'_1]), \\ \quad \vdots \\ \quad \overline{id_m}\triangleright((var_{poly}{:}sig_{poly}){\rightarrow}[\mathrm{it}\triangleright con_m \rightharpoonup con'_m])] \end{array}} \tag{238}$$

Rule 238: This rule handles recursive `val` bindings. As in Rule 237, we assume that the implicit scoping of explicit type variables has been made explicit by a prepass over the EL.

$$\frac{\begin{array}{c} \Gamma \vdash strdec_1 \rightsquigarrow mod_1 : sig_1 \\ var_1 \notin \mathrm{bound}(\Gamma) \\ \Gamma, lbl_1{}^*\triangleright var_1{:}sig_1 \vdash strdec_2 \rightsquigarrow mod_2 : sig_2 \end{array}}{\begin{array}{l} \Gamma \vdash strdec_1\ strdec_2 \rightsquigarrow \\ \quad [lbl_1{}^*\triangleright var_1{=}mod_1, lbl_2{}^*\triangleright var_2{=}mod_2] : \\ \quad [lbl_1{}^*\triangleright var_1{:}sig_1, lbl_2{}^*\triangleright var_2{:}sig_2] \end{array}} \tag{239}$$

Rule 239: Here we have a form of *semantic* concatenation, since syntactic concatenation could result in malformed signatures with duplicated labels (if EL identifiers are redefined). Additionally, some declarations (Rules 240 and 244) do not translate into explicit structure values.

$$\frac{\Gamma \vdash_{\mathrm{ctx}} \overline{longstrid_1} \rightsquigarrow path_1 : sig_1 \qquad \Gamma \vdash_{\mathrm{ctx}} \overline{longstrid_n} \rightsquigarrow path_n : sig_n}{\Gamma \vdash \texttt{open}\ longstrid_1\ \cdots\ longstrid_n \rightsquigarrow path : sig} \tag{240}$$

Rule 240: Though structures and datatypes both translate to structures, they have different namespaces so that *longstrid* can never specify a datatype.

$$\frac{\Gamma \vdash ty \rightsquigarrow con : \Omega \qquad var \notin \mathrm{bound}(\Gamma)}{\begin{array}{l}\Gamma \vdash \texttt{exception}\ id\ \texttt{of}\ ty \rightsquigarrow \\ \quad [\overline{id}\triangleright[lbl\triangleright var{=}\mathsf{new\_stamp}[con], \mathrm{mk}\triangleright var\#\mathrm{mk}, \mathrm{km}\triangleright var\#\mathrm{km}]] : \\ \quad [\overline{id}\triangleright[\mathrm{it}\triangleright var{:}\{\mathrm{mk}\triangleright con{\to}\mathsf{Any}, \mathrm{km}\triangleright\mathsf{Any}{\rightharpoonup}con\}, \\ \qquad \mathrm{mk}\triangleright con{\to}\mathsf{Any}, \mathrm{km}\triangleright\mathsf{Any}{\rightharpoonup}con]]\end{array}} \tag{241}$$

Rule 241: Note that the type of km is partial, whereas the type of mk is total.

$$\frac{\Gamma \vdash ty \rightsquigarrow \mathsf{Unit} : \Omega \qquad var \notin \mathrm{bound}(\Gamma)}{\begin{array}{l}\Gamma \vdash \texttt{exception}\ id \rightsquigarrow \\ \quad [\overline{id}\triangleright[\mathrm{it}\triangleright var{=}\mathsf{new\_stamp}[\mathsf{Unit}], \mathrm{mk}\triangleright(var\#\mathrm{mk})\,\{\}, \mathrm{km}\triangleright var\#\mathrm{km}]] : \\ \quad [\overline{id}\triangleright[\mathrm{it}\triangleright var{:}\{\mathrm{mk}\triangleright\mathsf{Unit}{\to}\mathsf{Any}, \mathrm{km}\triangleright\mathsf{Any}{\rightharpoonup}\mathsf{Unit}\}, \\ \qquad \mathrm{mk}\triangleright\mathsf{Any}, \mathrm{km}\triangleright\mathsf{Any}{\rightharpoonup}\mathsf{Unit}]]\end{array}} \tag{242}$$

Rule 242: Nullary exceptions require special treatment, alas.

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{ctx}} \overline{longid} \rightsquigarrow path : sig \\ \Gamma \vdash path.\mathrm{mk} : con_{\mathrm{mk}} \qquad \Gamma \vdash path.\mathrm{km} : \mathsf{Any}{\rightharpoonup}con\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{exception}\ id\ \texttt{=}\ longid \rightsquigarrow \\ \quad [\overline{id}\triangleright[\mathrm{mk}\triangleright path.\mathrm{mk}, \mathrm{km}\triangleright path.\mathrm{km}]] : \\ \quad [\overline{id}\triangleright[\mathrm{mk}\triangleright con_{\mathrm{mk}}, \mathrm{km}\triangleright\mathsf{Any}{\rightharpoonup}con]]\end{array}} \tag{243}$$

Rule 243: We check that *longid* corresponds to an exception constructor (and not a datatype constructor) by looking at the type of the km component.

$$\frac{\begin{array}{c}var_1 \notin \mathrm{bound}(\Gamma) \\ \Gamma \vdash strdec_1 \rightsquigarrow mod_1 : sig_1 \\ \Gamma, lbl_1{}^*\triangleright var_1{:}sig_1 \vdash strdec_2 \rightsquigarrow mod_2 : sig_2 \\ \Gamma, var_1{:}sig_1 \vdash sig_2 \leq sig : \mathsf{Sig} \qquad \Gamma \vdash sig : \mathsf{Sig}\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{local}\ strdec_1\ \texttt{in}\ strdec_2\ \texttt{end} \rightsquigarrow \\ \quad [lbl_1{}^*\triangleright var_1{=}mod_1, lbl_2\triangleright var_2{=}mod_2{:}sig].lbl_2 : sig\end{array}} \tag{244}$$

Rule 244: We create a two-component structure containing all the bindings in both declarations, but do not expose the local bindings. Note that the visible bindings must be typable with respect to the ambient environment, which does not include abstract types defined locally.

$$\frac{\Gamma \vdash \mathit{tybind} \rightsquigarrow \mathit{mod} : \mathit{sig}}{\Gamma \vdash \texttt{type}\ \mathit{tybind} \rightsquigarrow \mathit{mod} : \mathit{sig}} \tag{245}$$

$$\frac{\Gamma \vdash \mathit{datbind} \rightsquigarrow \mathit{mod} : \mathit{sig}}{\Gamma \vdash \texttt{datatype}\ \mathit{datbind} \rightsquigarrow \mathit{mod} : \mathit{sig}} \tag{246}$$

$$\frac{\Gamma \vdash \mathit{strbind} \rightsquigarrow \mathit{mod} : \mathit{sig}}{\Gamma \vdash \texttt{structure}\ \mathit{strbind} \rightsquigarrow \mathit{mod} : \mathit{sig}} \tag{247}$$

$$\frac{\Gamma \vdash \mathit{funbind} \rightsquigarrow \mathit{mod} : \mathit{sig}}{\Gamma \vdash \texttt{functor}\ \mathit{funbind} \rightsquigarrow \mathit{mod} : \mathit{sig}} \tag{248}$$

Structure Expressions $\boxed{\Gamma \vdash \mathit{strexp} \rightsquigarrow \mathit{mod} : \mathit{sig}}$

$$\frac{\Gamma \vdash \mathit{strdec} \rightsquigarrow \mathit{mod} : \mathit{sig}}{\Gamma \vdash \texttt{struct}\ \mathit{strdec}\ \texttt{end} \rightsquigarrow \mathit{mod} : \mathit{sig}} \tag{249}$$

$$\frac{\Gamma \vdash_{\text{ctx}} \overline{\mathit{longstrid}} \rightsquigarrow \mathit{path} : \mathit{sig}}{\Gamma \vdash \mathit{longstrid} \rightsquigarrow \mathit{path} : \mathit{sig}} \tag{250}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\text{ctx}} \overline{\mathit{funid}} \rightsquigarrow \mathit{path} : ((\mathit{var}_1{:}\mathit{sig}_1) \rightharpoonup \mathit{sig}_2) \\ \Gamma \vdash \mathit{strexp} \rightsquigarrow \mathit{mod} : \mathit{sig} \\ \Gamma \vdash_{\text{sub}} \mathit{sig} \preceq \mathit{sig}_1 \rightsquigarrow (\lambda \mathit{var}{:}\mathit{sig}.\mathit{mod}_1) : ((\mathit{var}{:}\mathit{sig}) \rightarrow \mathit{sig}_1'') \\ \Gamma \vdash ((\mathit{var}_1{:}\mathit{sig}_1) \rightharpoonup \mathit{sig}_2) \leq (\mathit{sig}_1' \rightharpoonup \mathit{sig}_2') : \mathsf{Sig}\end{array}}{\Gamma \vdash \mathit{funid}\,(\mathit{strexp}) \rightsquigarrow \mathit{path}\,((\lambda \mathit{var}{:}\mathit{sig}.\mathit{mod}_1)\,\mathit{mod}) : \mathit{sig}_2'} \tag{251}$$

Rule 251:

- We insert an explicit coercion to make the argument structure (which has signature *sig*) match the domain signature of the functor (sig_1).
- EL polymorphic identifiers cannot be applied as functors in the EL for two reasons: the result of doing the lookup must have a partial functor type, and the name-spaces of functor identifiers and value identifiers are disjoint.‘

$$\frac{\begin{array}{c} var \notin \mathrm{bound}(\Gamma) \\ \Gamma \vdash strexp \rightsquigarrow mod : sig \qquad \Gamma \vdash sigexp \rightsquigarrow sig' : \mathsf{Sig} \\ \Gamma \vdash_{\mathrm{sub}} sig \preceq sig' \rightsquigarrow (\lambda var{:}sig.mod') : ((var{:}sig) \rightarrow sig'') \end{array}}{\Gamma \vdash strexp \ : \ sigexp \rightsquigarrow (\lambda var{:}sig.mod')\, mod : sig''} \tag{252}$$

Rule 252: As in SML, ascribing a signature to a structure using ":" hides components (this hiding being accomplished here via an explicit coercion), but allows the identity of the remaining type components to leak through. The rules for coercions ensure that sig'' maximizes propagation of type information.

$$\frac{\begin{array}{c} \Gamma \vdash strexp \rightsquigarrow mod : sig \\ \Gamma \vdash sigexp \rightsquigarrow sig' : \mathsf{Sig} \\ var \notin \mathrm{bound}(\Gamma) \\ \Gamma \vdash_{\mathrm{sub}} sig \preceq sig' \rightsquigarrow (\lambda var{:}sig.mod') : ((var{:}sig) \rightarrow sig'') \end{array}}{\Gamma \vdash strexp \ \texttt{:>}\ sigexp \rightsquigarrow (\lambda var{:}sig.(mod'{:}sig'))\, mod : sig'} \tag{253}$$

Rule 253: Ascribing a signature to a structure with `:>` not only hides components, but restricts information about types to that which appears in the signature.

$$\frac{\begin{array}{c} var_1 \notin \mathrm{bound}(\Gamma) \\ \Gamma \vdash strdec \rightsquigarrow mod_1 : sig_1 \\ \Gamma, lbl_1{}^* \triangleright var_1{:}sig_1 \vdash strexp \rightsquigarrow mod_2 : sig_2 \\ \Gamma, var_1{:}sig_1 \vdash sig_2 \leq sig : \mathsf{Sig} \qquad \Gamma \vdash sig : \mathsf{Sig} \end{array}}{\begin{array}{l} \Gamma \vdash \texttt{let}\ strdec\ \texttt{in}\ strexp\ \texttt{end} \rightsquigarrow \\ \qquad [lbl_1{}^* \triangleright var_1{=}mod_1, lbl_2 \triangleright var_2{=}mod_2{:}sig].lbl_2 : sig \end{array}} \tag{254}$$

Rule 254: We create a two-component structure which includes the local module binding, but do not expose this local module. Note that the visible structure must be typable with respect to the ambient environment, which does not include abstract types in the local module.

Structure Binding

$\boxed{\Gamma \vdash strbind \rightsquigarrow mod : sig}$

$$\frac{\Gamma \vdash strexp_1 \rightsquigarrow mod_1 : sig_1 \qquad \langle \cdots \qquad \Gamma \vdash strexp_n \rightsquigarrow mod_n : sig_n \rangle}{\begin{array}{l} \Gamma \vdash strid_1\ \texttt{=}\ strexp_1\ \langle \texttt{and}\ \cdots\ \texttt{and}\ strid_n\ \texttt{=}\ strexp_n \rangle \rightsquigarrow \\ \qquad [\overline{strid_1} \triangleright mod_1 \langle, \cdots, \overline{strid_n} \triangleright mod_n \rangle] : [\overline{strid_1} \triangleright sig_1 \langle, \cdots, \overline{strid_n} \triangleright sig_n \rangle] \end{array}} \tag{255}$$

Functor Binding

$\boxed{\Gamma \vdash \mathit{funbind} \rightsquigarrow \mathit{mod} : \mathit{sig}}$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{sigexp}_1 \rightsquigarrow \mathit{sig}_1 : \mathsf{Sig} \qquad \begin{array}{c}\mathit{var} \notin \mathrm{bound}(\Gamma) \\ \Gamma, \overline{\mathit{sigid}_1} \triangleright \mathit{var}{:}\mathit{sig}_1 \vdash \mathit{strexp}_1 \rightsquigarrow \mathit{mod}_1 : \mathit{sig}'_1\end{array} \\ \langle \vdots \rangle \\ \langle \Gamma \vdash \mathit{sigexp}_n \rightsquigarrow \mathit{sig}_n : \mathsf{Sig} \qquad \Gamma, \overline{\mathit{sigid}_n} \triangleright \mathit{var}{:}\mathit{sig}_n \vdash \mathit{strexp}_n \rightsquigarrow \mathit{mod}_n : \mathit{sig}'_n \rangle\end{array}}{\begin{array}{l}\Gamma \vdash \mathit{id}_1\texttt{(}\mathit{sigid}_1 : \mathit{sigexp}_1\texttt{)} = \mathit{strexp}_1 \\ \quad \langle \texttt{and} \cdots \texttt{and}\ \mathit{id}_n\texttt{(}\mathit{sigid}_n : \mathit{sigexp}_1\texttt{)} = \mathit{strexp}_n \rangle \rightsquigarrow \\ \quad \overline{\mathit{funid}_1} \triangleright (\lambda \mathit{var}{:}\mathit{sig}_1.\mathit{mod}_1)\langle, \cdots, \overline{\mathit{funid}_n} \triangleright (\lambda \mathit{var}{:}\mathit{sig}_n.\mathit{mod}_n)\rangle : \\ \quad \overline{\mathit{funid}_1} \triangleright ((\mathit{var}{:}\mathit{sig}_1) \rightharpoonup \mathit{sig}'_1)\langle, \cdots, \overline{\mathit{funid}_n} \triangleright ((\mathit{var}{:}\mathit{sig}_n) \rightharpoonup \mathit{sig}'_n)\rangle\end{array}} \tag{256}$$

Rule 256: All functors defined explicitly in the EL have partial arrows.

Specifications

$\boxed{\Gamma \vdash \mathit{spec} \rightsquigarrow [\mathit{sdecs}] : \mathsf{Sig}}$

$$\frac{\begin{array}{c}\mathrm{FTV}(\mathit{ty}) = \emptyset \\ \Gamma \vdash \mathit{ty} \rightsquigarrow \mathit{con} : \Omega\end{array}}{\Gamma \vdash \texttt{val}\ \mathit{id}\ :\ \mathit{ty} \rightsquigarrow [\overline{\mathit{id}} \triangleright \mathit{con}] : \mathsf{Sig}} \tag{257}$$

$$\frac{\begin{array}{c}\mathrm{FTV}(\mathit{ty}) = \{\mathit{tyvar}_1, \cdots, \mathit{tyvar}_n\} \neq \emptyset \\ \mathit{sig}_{\mathit{poly}} = [\overline{\mathit{tyvar}_1} \triangleright \Omega, \cdots, \overline{\mathit{tyvar}_n} \triangleright \Omega] \\ \Gamma, \mathit{lbl}^* \triangleright \mathit{var}{:}\mathit{sig}_{\mathit{poly}} \vdash \mathit{ty} \rightsquigarrow \mathit{con} : \Omega\end{array}}{\Gamma \vdash \texttt{val}\ \mathit{id}\ :\ \mathit{ty} \rightsquigarrow [\overline{\mathit{id}} \triangleright ((\mathit{var}{:}\mathit{sig}_{\mathit{poly}}) \rightarrow [\mathrm{it} \triangleright \mathit{con}])] : \mathsf{Sig}} \tag{258}$$

$$\frac{\Gamma \vdash \mathit{typdesc} \rightsquigarrow \mathit{sig} : \mathsf{Sig}}{\Gamma \vdash \texttt{type}\ \mathit{typdesc} \rightsquigarrow \mathit{sig} : \mathsf{Sig}} \tag{259}$$

$$\frac{\Gamma \vdash \mathit{ty} \rightsquigarrow \mathit{con} : \Omega}{\begin{array}{l}\Gamma \vdash \texttt{exception}\ \mathit{id}\ \texttt{of}\ \mathit{ty} \rightsquigarrow \\ \quad [\overline{\mathit{id}} \triangleright [\mathrm{mk} \triangleright \mathit{con} \rightarrow \mathsf{Any}, \mathrm{km} \triangleright \mathsf{Any} \rightharpoonup \mathit{con}]] : \mathsf{Sig}\end{array}} \tag{260}$$

$$\frac{}{\Gamma \vdash \texttt{exception}\ \mathit{id} \rightsquigarrow [\overline{\mathit{id}} \triangleright [\mathrm{mk} \triangleright \mathsf{Any}, \mathrm{km} \triangleright \mathsf{Any} \rightharpoonup \mathsf{Unit}]] : \mathsf{Sig}} \tag{261}$$

$$\frac{\Gamma \vdash \mathit{datbind} \rightsquigarrow \mathit{mod} : \mathit{sig}}{\Gamma \vdash \texttt{datatype}\ \mathit{datbind} \rightsquigarrow \mathit{sig} : \mathsf{Sig}} \tag{262}$$

$$\frac{\Gamma \vdash sigexp \rightsquigarrow sig : \mathsf{Sig}}{\Gamma \vdash \texttt{structure}\ strid\ :\ sigexp \rightsquigarrow [\overline{strid} \triangleright sig] : \mathsf{Sig}} \tag{263}$$

$$\frac{\begin{array}{c}\Gamma \vdash sigexp \rightsquigarrow sig : \mathsf{Sig} \\ var \notin \mathrm{bound}(\Gamma) \\ \Gamma, \overline{strid} \triangleright var{:}sig \vdash sigexp' \rightsquigarrow sig' : \mathsf{Sig}\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{functor}\ funid\ \texttt{(}strid\ :\ sigexp\texttt{)}\ :\ sigexp' \rightsquigarrow \\ \quad [\overline{strid} \triangleright (var{:}sig \rightharpoonup sig')] : \mathsf{Sig}\end{array}} \tag{264}$$

$$\frac{\Gamma \vdash sigexp \rightsquigarrow sig : \mathsf{Sig}}{\Gamma \vdash \texttt{include}\ sigexp \rightsquigarrow sig : \mathsf{Sig}} \tag{265}$$

$$\frac{\begin{array}{c}\Gamma \vdash spec_1 \rightsquigarrow [sdecs_1] : \mathsf{Sig} \qquad \Gamma, sdecs \vdash spec_2 \rightsquigarrow [sdecs_2] : \mathsf{Sig} \\ \Gamma \vdash [sdecs_1, sdecs_2] : \mathsf{Sig}\end{array}}{\Gamma \vdash spec_1\ spec_2 \rightsquigarrow [sdecs_1, sdecs_2] : \mathsf{Sig}} \tag{266}$$

Rule 266: We disallow redeclaring EL identifiers in a signature. In the presence of `include`, we cannot syntactically restrict the EL to guarantee the syntactic concatenation of $sdecs_1$ and $sdecs_2$ will be well-formed—hence we check here.

$$\frac{\begin{array}{c}\Gamma \vdash spec \rightsquigarrow sig : \mathsf{Sig} \\ var \notin \mathrm{bound}(\Gamma) \\ \Gamma, var{:}sig \vdash_{\mathsf{sig}} \overline{longid_1} \rightsquigarrow lbls_1 : knd \\ \Gamma, var{:}sig \vdash_{\mathsf{sig}} \overline{longid_2} \rightsquigarrow lbls_2 : knd \\ \Gamma, var{:}sig \vdash var.lbls_1 \equiv var.lbls_1' : knd \\ \Gamma, var{:}sig \vdash var.lbls_2 \equiv var.lbls_2' : knd \\ \left\{\begin{array}{c} sig \vdash_{\mathsf{sh}} lbls_1' := lbls_2' : knd \rightsquigarrow sig' : \mathsf{Sig} \\ \text{or} \\ sig \vdash_{\mathsf{sh}} lbls_2' := lbls_1' : knd \rightsquigarrow sig' : \mathsf{Sig} \end{array}\right\}\end{array}}{\Gamma \vdash spec\ \texttt{sharing type}\ longid_1\ \texttt{=}\ longid_2 \rightsquigarrow sig' : \mathsf{Sig}} \tag{267}$$

Rule 267: A type component in a signature is considered abstract, and hence eligible to appear in a sharing constraint, if it is equivalent to an opaque type (type component that is not a type abbreviation) in the signature.

Here we "guess" the two opaque types to which the given components are equivalent, and patch the signature such that the opaque type with the smaller scope becomes a type abbreviation for the other opaque type.

Signature Expressions $\boxed{\Gamma \vdash sigexp \rightsquigarrow sig : \mathsf{Sig}}$

$$\frac{\Gamma \vdash spec \rightsquigarrow sig : \mathsf{Sig}}{\Gamma \vdash \texttt{sig}\ spec\ \texttt{end} \rightsquigarrow sig : \mathsf{Sig}} \tag{268}$$

$$\frac{\Gamma \vdash_{\text{ctx}} \overline{sigid} \rightsquigarrow sig : \mathsf{Sig}}{\Gamma \vdash sigid \rightsquigarrow sig : \mathsf{Sig}} \tag{269}$$

$$\frac{\begin{array}{c} var, var_1, \cdots, var_n \notin \text{bound}(\Gamma) \\ \Gamma \vdash sigexp \rightsquigarrow sig : \mathsf{Sig} \\ \Gamma\langle, \overline{tyvar_1} \triangleright var_1{:}\Omega, \cdots, \overline{tyvar_n} \triangleright var_n{:}\Omega\rangle \vdash ty \rightsquigarrow con : \Omega \\ \Gamma, var{:}sig \vdash_{\overline{\text{sig}}} \overline{longtycon} \rightsquigarrow lbls : \langle \Omega^n \Rightarrow \rangle\, \Omega \\ \Gamma, var{:}sig \vdash var.lbls \equiv var.lbls' : \langle \Omega^n \Rightarrow \rangle\, \Omega \\ sig \vdash_{\text{wt}} lbls' := \langle \lambda\, (var_1, \cdots, var_n).\rangle con : \langle \Omega^n \Rightarrow \rangle\, \Omega \rightsquigarrow sig' : \mathsf{Sig} \end{array}}{\Gamma \vdash sigexp\ \texttt{where type}\ \langle (tyvar_1\texttt{,}\cdots\texttt{,}tyvar_n)\rangle\ longtycon\ \texttt{=}\ ty \rightsquigarrow sig' : \mathsf{Sig}} \tag{270}$$

Rule 270: This rule uses ambient scope for the EL type expression ty.

$$\frac{\begin{array}{c} var, var_1, \cdots, var_n \notin \text{bound}(\Gamma) \\ \Gamma \vdash sigexp \rightsquigarrow sig : \mathsf{Sig} \\ \Gamma, lbl^* \triangleright var{:}sig\langle, \overline{tyvar_1} \triangleright var_1{:}\Omega, \cdots, \overline{tyvar_n} \triangleright var_n{:}\Omega\rangle \vdash ty \rightsquigarrow con : \Omega \\ \Gamma, var{:}sig \vdash_{\overline{\text{sig}}} \overline{longtycon} \rightsquigarrow lbls : \langle \Omega^n \Rightarrow \rangle\, \Omega \\ \Gamma, var{:}sig \vdash var.lbls \equiv var.lbls' : \langle \Omega^n \Rightarrow \rangle\, \Omega \\ sig \vdash_{\text{wt}} lbls' := \langle \lambda\, (var_1, \cdots, var_n).\rangle con : \langle \Omega^n \Rightarrow \rangle\, \Omega \rightsquigarrow sig' : \mathsf{Sig} \\ \Gamma, var{:}sig \vdash sig' \equiv sig'' : \mathsf{Sig} \\ \Gamma \vdash sig'' : \mathsf{Sig} \end{array}}{\Gamma \vdash sigexp\ \texttt{where type}\ \langle (tyvar_1\texttt{,}\cdots\texttt{,}tyvar_n)\rangle\ longtycon\ \texttt{=}\ \texttt{own}\ ty \rightsquigarrow sig' : \mathsf{Sig}} \tag{271}$$

Rule 271: This rule patches signatures using internal scope for the type expression ty. As such, it may require reordering the signature components in order to satisfy scoping requirements.

Top Level

$\boxed{\Gamma \vdash topdec \rightsquigarrow tbnds : tdecs}$

$$\frac{\Gamma \vdash sigexp_1 \rightsquigarrow sig_1 : \mathsf{Sig} \qquad \langle \cdots \qquad \Gamma \vdash sigexp_n \rightsquigarrow sig_n : \mathsf{Sig} \rangle}{\begin{array}{l}\Gamma \vdash \texttt{signature}\ sigid_1\ \texttt{=}\ sigexp_1\ \langle \texttt{and}\ \cdots\ \texttt{and}\ sigid_n\ \texttt{=}\ sigexp_n \rangle \rightsquigarrow \\ \quad \overline{sigid_1} \triangleright sig_1 \langle , \cdots , \overline{sigid_n} \triangleright sig_n \rangle : \\ \quad \overline{sigid_1} \triangleright \mathsf{Sig} = sig_1 \langle , \cdots , \overline{sigid_n} \triangleright \mathsf{Sig} = sig_n \rangle\end{array}} \tag{272}$$

$$\frac{\Gamma \vdash strbind \rightsquigarrow [sbnds] : [sdecs]}{\Gamma \vdash \texttt{structure}\ strbind \rightsquigarrow sbnds : sdecs} \tag{273}$$

$$\frac{\Gamma \vdash funbind \rightsquigarrow [sbnds] : [sdecs]}{\Gamma \vdash \texttt{functor}\ funbind \rightsquigarrow sbnds : sdecs} \tag{274}$$

$$\frac{\begin{array}{c}\Gamma \vdash topdec_1 \rightsquigarrow tbnds_1 : tdecs_1 \qquad \Gamma, tdecs_1 \vdash topdec_2 \rightsquigarrow tbnds_2 : tdecs_2 \\ \Gamma \vdash tdecs_1, tdecs_2\ \text{ok}\end{array}}{\Gamma \vdash topdec_1\ topdec_2 \rightsquigarrow tbnds_1, tbnds_2 : tdecs_1, tdecs_2} \tag{275}$$

Rule 275: We check that there are no duplicate definitions at the top level.

Type Expressions

$\boxed{\Gamma \vdash ty \rightsquigarrow con : \Omega}$

$$\frac{}{\Gamma \vdash tyvar \rightsquigarrow \overline{tyvar} : \Omega} \tag{276}$$

$$\frac{\Gamma \vdash ty_1 \rightsquigarrow con_1 : \Omega \qquad \cdots \qquad \Gamma \vdash ty_n \rightsquigarrow con_n : \Omega}{\Gamma \vdash \{lab_1 : ty_1, \cdots, lab_n : ty_n\} \rightsquigarrow \{\overline{lab_1} \triangleright con_1, \cdots, \overline{lab_n} \triangleright con_n\} : \Omega} \tag{277}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\text{ctx}} \overline{longtycon} \rightsquigarrow path : \langle \Omega^n \Rightarrow \rangle\, \Omega \\ \langle \Gamma \vdash ty_1 \rightsquigarrow con_1 : \Omega \quad \cdots \quad \Gamma \vdash ty_n \rightsquigarrow con_n : \Omega \rangle\end{array}}{\Gamma \vdash \langle (ty_1, \cdots, ty_n) \rangle\ longtycon \rightsquigarrow path \langle (con_1, \cdots, con_n) \rangle : \Omega} \tag{278}$$

$$\frac{\Gamma \vdash ty \rightsquigarrow con : \Omega \qquad \Gamma \vdash ty' \rightsquigarrow con' : \Omega}{\Gamma \vdash ty\ \texttt{->}\ ty' \rightsquigarrow con \rightharpoonup con' : \Omega} \tag{279}$$

Rule 279: There is no way to directly denote a total ($\rightarrow$) type in the external language.

Type Definitions

$\boxed{\Gamma \vdash tybind \rightsquigarrow [sbnds] : [sdecs]}$

$$\frac{\begin{array}{c}\langle var_1, \cdots, var_n \notin \mathrm{bound}(\Gamma)\rangle \\ \Gamma, \langle \overline{tyvar_1} \triangleright var_1{:}\Omega, \cdots, \overline{tyvar_n} \triangleright var_n{:}\Omega\rangle \vdash ty \rightsquigarrow con' : \Omega \\ con := \langle \lambda\,(var_1, \cdots, var_n).\rangle con \\ \langle\langle \Gamma \vdash tybind \rightsquigarrow [sbnds] : [sdecs]\rangle\rangle \qquad \langle\langle var \notin \mathrm{FV} sdecs\rangle\rangle\end{array}}{\begin{array}{l}\Gamma \vdash \langle (tyvar_1, \cdots, tyvar_n)\rangle\ tycon \texttt{ = } ty\ \langle\langle \texttt{and } tybind\rangle\rangle \rightsquigarrow \\ \quad [\overline{tycon} \triangleright var{=}con\langle\langle, sbnds\rangle\rangle] : \\ \quad [\overline{tycon} \triangleright var{:}\langle \Omega^n \Rightarrow \rangle\, \Omega{=}con\langle\langle, sdecs\rangle\rangle]\end{array}} \tag{280}$$

Type Descriptions

$\boxed{\Gamma \vdash typdesc \rightsquigarrow [sdecs] : \mathsf{Sig}}$

$$\frac{\langle\langle \Gamma \vdash typdesc \rightsquigarrow [sdecs] : \mathsf{Sig}\rangle\rangle}{\begin{array}{l}\Gamma \vdash \texttt{type}\ \langle (tyvar_1, \cdots, tyvar_n)\rangle\ id\ \langle\langle\ \texttt{and } typdesc\rangle\rangle \rightsquigarrow \\ \quad [\overline{id} \triangleright \langle \Omega^n \Rightarrow \rangle\, \Omega\langle\langle, sdecs\rangle\rangle] : \mathsf{Sig}\end{array}} \tag{281}$$

$$\frac{\begin{array}{c}\langle var_1, \cdots, var_n \notin \mathrm{bound}(\Gamma)\rangle \\ \Gamma, \langle \overline{tyvar_1} \triangleright var_1{:}\Omega, \cdots, \overline{tyvar_n} \triangleright var_n{:}\Omega\rangle \vdash ty \rightsquigarrow con' : \Omega \\ con := \langle \lambda\,(var_1, \cdots, var_n).\rangle con \\ \langle\langle \Gamma \vdash typdesc \rightsquigarrow [sdecs] : \mathsf{Sig}\rangle\rangle \qquad \langle\langle var \notin \mathrm{FV} sdecs\rangle\rangle\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{type}\ \langle (tyvar_1, \cdots, tyvar_n)\rangle\ id\ \texttt{=}\ ty\ \langle\langle\ \texttt{and } typdesc\rangle\rangle \rightsquigarrow \\ \quad [\overline{id} \triangleright \langle \Omega^n \Rightarrow \rangle\, \Omega{=}con\langle\langle, sdecs\rangle\rangle] : \mathsf{Sig}\end{array}} \tag{282}$$

Datatype Definitions

$\boxed{\Gamma \vdash datbind \rightsquigarrow mod : sig}$

To make the definition somewhat manageable, we define the necessary many types and expressions in steps:

- $tyvar_1, \cdots, tyvar_k$ is the union of the explicit type variables scoped in any of the datatype definitions;
- con_i^{sum} is roughly the sum type representing the i^{th} datatype, with free variables $var_1^{poly}, \cdots, var_k^{poly}$ representing the type variables due to polymorphism, and $var_1^{dt}, \cdots, var_p^{dt}$ representing the final implementation types for the datatypes.
- The implementation type for the i^{th} datatype is $\mu_i\,[con^{all}]$, which still contains $var_1^{poly}, \cdots, var_k^{poly}$ free.
- var^{all} (appearing in the translation) is a constructor mapping a tuple of types (the polymorphic instantiation types) to a tuple of types (the datatype implementations).
- con_i^{dt} is the implementation of the i^{th} datatype with respect to a *structure* of types var_{poly}.
- con_{ij}'' is the domain of the constructor id_{ij}, if the constructor carries a value.
- exp_{ij}^{mk} is the translation of the constructor id_{ij}, and con_{ij}^{mk} is its type, which is always total. These contain free variables var_{poly} and var^{all}.
- exp_{ij}^{km} is the translation of the deconstructor for id_{ij}, and con_{ij}^{km} is its type. The type is partial unless there is exactly one constructor in the datatype. These contain free variables var_{poly} and var^{all}.
- exp_i^{case} is the non-destructuring case statement for the i^{th} datatype, and con_i^{case} is its type. These contain free variables var_{poly} and var^{all}.

$$\{tyvar_1, \cdots, tyvar_k\} = \{tyvar_{11}, \cdots, tyvar_{1n_1}\} \cup \cdots \cup \{tyvar_{p1}, \cdots, tyvar_{pn_p}\}$$
$$sig_{poly} := [\overline{tyvar_1} \triangleright var_1^{poly}{:}\Omega, \cdots, \overline{tyvar_k} \triangleright var_k^{poly}{:}\Omega]$$
$$sig_{poly}^{+} := [\overline{tyvar_1} \triangleright var_1^{poly}{:}\Omega, \cdots, \overline{tyvar_k} \triangleright var_k^{poly}{:}\Omega, lbl \triangleright \Omega{:}]$$

$$\Gamma' := \Gamma, lbl_{poly}{}^{*} \triangleright var_{poly}{:}sig_{poly}, \overline{tyvar_1} \triangleright var_1^{poly}{:}\Omega, \cdots, \overline{tyvar_k} \triangleright var_k^{poly}{:}\Omega,$$
$$\overline{tycon_1} \triangleright var_1^{ty}{:}\Omega^k \Rightarrow \Omega, \cdots, \overline{tycon_p} \triangleright var_p^{ty}{:}\Omega^k \Rightarrow \Omega$$
$$\langle con'_{ij} := [var_1^{dt}/var_1^{ty}[\cdots]] \cdots [var_p^{dt}/var_p^{ty}[\cdots]] con_{ij} \rangle_{ij}$$
$$\langle \Gamma' \vdash ty_{ij} \rightsquigarrow con'_{ij} : \Omega \rangle_{ij}$$
$$con_i^{sum} := \Sigma \left(\left\{ \begin{array}{c} \mathsf{Unit} \\ \text{or} \\ con'_{i1} \end{array} \right\}_{i1}, \cdots, \left\{ \begin{array}{c} \mathsf{Unit} \\ \text{or} \\ con'_{im_i} \end{array} \right\}_{im_i} \right)$$
$$con^{all} := \lambda\,(var_1^{dt}, \cdots, var_p^{dt}).(con_1^{sum}, \cdots, con_p^{sum})$$
$$con_i^{dt} := (var^{all}[(var_{poly}.\overline{tyvar_1}, \cdots, var_{poly}.\overline{tyvar_k})])\#i$$
$$\langle con''_{ij} := [con_1^{dt}/var_1^{dt}] \cdots [con_p^{dt}/var_p^{dt}] con'_{ij} \rangle_{ij}$$

$$exp_{ij}^{\mathrm{mk}} := \left\{ \begin{array}{c} \mathsf{roll}_i^{con_i^{dt}}\ (\mathsf{inj}_j^{\Sigma\left(con''_{i1}, \cdots, con''_{im_i}\right)}\ \{\}) \\ \text{or} \\ \lambda\, var{:}con''_{ij}.\mathsf{roll}_i^{con_i^{dt}}\ (\mathsf{inj}_j^{\Sigma\left(con''_{i1}, \cdots, con''_{im_i}\right)}\ var) \end{array} \right\}_{ij}$$
$$con_{ij}^{\mathrm{mk}} := \left\{ \begin{array}{c} con_i^{dt} \\ \text{or} \\ con''_{ij} \rightarrow con_i^{dt} \end{array} \right\}_{ij}$$
$$exp_{ij}^{\mathrm{km}} := \lambda\, var{:}con_i^{dt}.\mathsf{proj}_j\ (\mathsf{unroll}_i^{con_i^{dt}}\ var)$$
$$con_{ij}^{\mathrm{km}} := \left\{ \begin{array}{ll} con_i^{dt} \rightarrow con''_{ij} & \text{if } m_i = 1; \\ con_i^{dt} \rightharpoonup con''_{ij} & \text{otherwise.} \end{array} \right.$$
$$exp_i^{\mathrm{case}} := \lambda\, var{:}\{1 \triangleright con_i^{dt} \rightharpoonup var_{poly}.lbl, \cdots, m_i \triangleright con_i^{dt} \rightharpoonup var_{poly}.lbl\}.$$
$$\lambda\, var'{:}con_i^{dt}.\mathsf{case}^{\Sigma\left(con''_{i1}, \cdots, con''_{im_i}\right)}\ (var\#1, \cdots, var\#m_i)\ \mathsf{of}\ \mathsf{unroll}_i^{con_i^{dt}}\ var'\ \mathsf{end}$$
$$con_i^{\mathrm{case}} := \{1 \triangleright con_i^{dt} \rightharpoonup var_{poly}.lbl, \cdots, m_i \triangleright con_i^{dt} \rightharpoonup var_{poly}.lbl\} \rightarrow con_i^{dt} \rightharpoonup var_{poly}.lbl$$

$$\Gamma \vdash \ (tyvar_{11}, \cdots, tyvar_{1n_1})\ tycon_1 \texttt{=} id_{11} \left\{ \begin{array}{c} \text{or} \\ \texttt{of}\ ty_{11} \end{array} \right\}_{11} \texttt{|} \cdots \texttt{|}\, id_{1m_1} \left\{ \begin{array}{c} \text{or} \\ \texttt{of}\ ty_{1m_1} \end{array} \right\}_{1m_1}$$
$$\texttt{and}\ \cdots \texttt{and}$$
$$(tyvar_{p1}, \cdots, tyvar_{pn_p})\ tycon_p \texttt{=} id_{p1} \left\{ \begin{array}{c} \text{or} \\ \texttt{of}\ ty_{p1} \end{array} \right\}_{p1} \texttt{|} \cdots \texttt{|}\, id_{pm_p} \left\{ \begin{array}{c} \text{or} \\ \texttt{of}\ ty_{pm_p} \end{array} \right\}_{pm_p} \rightsquigarrow \cdots$$
(283)

$$
\begin{array}{l}
\Big[lbl \triangleright var^{\,all} = \lambda\,(var_1^{\,poly}, \cdots, var_k^{\,poly}).(\mu_1\,[con^{\,all}], \cdots, \mu_p\,[con^{\,all}]), \\
\quad \overline{tycon_1}^{\,*} \triangleright \Big[\overline{tycon_1} \triangleright \lambda\,(var_1^{\,poly}, \cdots, var_k^{\,poly}).(var^{\,all}\,[(var_1^{\,poly}, \cdots, var_k^{\,poly})])\#1, \\
\qquad \overline{id_{11}} \triangleright [\mathrm{mk} \triangleright (\lambda var_{\,poly} : sig_{\,poly}.[\mathrm{it} \triangleright exp_{11}^{\mathrm{mk}}]), \\
\qquad\qquad \mathrm{km} \triangleright (\lambda var_{\,poly} : sig_{\,poly}.[\mathrm{it} \triangleright exp_{11}^{\mathrm{km}}])], \cdots, \\
\qquad \overline{id_{1m_1}} \triangleright [\mathrm{mk} \triangleright (\lambda var_{\,poly} : sig_{\,poly}.[\mathrm{it} \triangleright exp_{1m_1}^{\mathrm{mk}}]), \\
\qquad\qquad \mathrm{km} \triangleright (\lambda var_{\,poly} : sig_{\,poly}.[\mathrm{it} \triangleright exp_{1m_1}^{\mathrm{km}}])], \\
\qquad \mathrm{case} \triangleright (\lambda var_{\,poly} : sig_{\,poly}^{+}.[\mathrm{it} \triangleright exp_1^{\mathrm{case}}])\Big] \\
\vdots \\
\quad \overline{tycon_p}^{\,*} \triangleright \Big[\overline{tycon_p} \triangleright \lambda\,(var_1^{\,poly}, \cdots, var_k^{\,poly}).(var^{\,all}\,[(var_1^{\,poly}, \cdots, var_k^{\,poly})])\#1, \\
\qquad \overline{id_{p1}} \triangleright [\mathrm{mk} \triangleright (\lambda var_{\,poly} : sig_{\,poly}.[\mathrm{it} \triangleright exp_{p1}^{\mathrm{mk}}]), \\
\qquad\qquad \mathrm{km} \triangleright (\lambda var_{\,poly} : sig_{\,poly}.[\mathrm{it} \triangleright exp_{p1}^{\mathrm{km}}])], \cdots, \\
\qquad \overline{id_{pm_p}} \triangleright [\mathrm{mk} \triangleright (\lambda var_{\,poly} : sig_{\,poly}.[\mathrm{it} \triangleright exp_{pm_p}^{\mathrm{mk}}]), \\
\qquad\qquad \mathrm{km} \triangleright (\lambda var_{\,poly} : sig_{\,poly}.[\mathrm{it} \triangleright exp_{pm_p}^{\mathrm{km}}])], \\
\qquad \mathrm{case} \triangleright (\lambda var_{\,poly} : sig_{\,poly}^{+}.[\mathrm{it} \triangleright exp_p^{\mathrm{case}}])\Big] \\
\Big] : \\
\\
\Big[lbl \triangleright var^{\,all} : \Omega^k \Rightarrow \Omega^p, \\
\quad \overline{tycon_1}^{\,*} \triangleright \Big[\overline{tycon_1} \triangleright \Omega^k \Rightarrow \Omega \\
\qquad\qquad = \lambda\,(var_1^{\,poly}, \cdots, var_k^{\,poly}).(var^{\,all}\,[(var_1^{\,poly}, \cdots, var_k^{\,poly})])\#1, \\
\qquad \overline{id_{11}} \triangleright [\mathrm{mk} \triangleright ((var_{\,poly} : sig_{\,poly}) \rightarrow [\mathrm{it} \triangleright con_{11}^{\mathrm{mk}}]), \\
\qquad\qquad \mathrm{km} \triangleright ((var_{\,poly} : sig_{\,poly}) \rightarrow [\mathrm{it} \triangleright con_{11}^{\mathrm{km}}])], \cdots, \\
\qquad \overline{id_{1m_1}} \triangleright [\mathrm{mk} \triangleright ((var_{\,poly} : sig_{\,poly}) \rightarrow [\mathrm{it} \triangleright con_{1m_1}^{\mathrm{mk}}]), \\
\qquad\qquad \mathrm{km} \triangleright ((var_{\,poly} : sig_{\,poly}) \rightarrow [\mathrm{it} \triangleright con_{1m_1}^{\mathrm{km}}])] \\
\qquad \mathrm{case} \triangleright (\lambda var_{\,poly} : sig_{\,poly}^{+}.[\mathrm{it} \triangleright con_1^{\mathrm{case}}])\Big] \\
\vdots \\
\quad \overline{tycon_p}^{\,*} \triangleright \Big[\overline{tycon_1} \triangleright \Omega^k \Rightarrow \Omega \\
\qquad\qquad = \lambda\,(var_1^{\,poly}, \cdots, var_k^{\,poly}).(var^{\,all}\,[(var_1^{\,poly}, \cdots, var_k^{\,poly})])\#1, \\
\qquad \overline{id_{p1}} \triangleright [\mathrm{mk} \triangleright ((var_{\,poly} : sig_{\,poly}) \rightarrow [\mathrm{it} \triangleright con_{p1}^{\mathrm{mk}}]), \\
\qquad\qquad \mathrm{km} \triangleright ((var_{\,poly} : sig_{\,poly}) \rightarrow [\mathrm{it} \triangleright con_{p1}^{\mathrm{km}}])], \cdots, \\
\qquad \overline{id_{pm_p}} \triangleright [\mathrm{mk} \triangleright ((var_{\,poly} : sig_{\,poly}) \rightarrow [\mathrm{it} \triangleright con_{pm_p}^{\mathrm{mk}}]), \\
\qquad\qquad \mathrm{km} \triangleright ((var_{\,poly} : sig_{\,poly}) \rightarrow [\mathrm{it} \triangleright con_{pm_p}^{\mathrm{km}}])] \\
\qquad \mathrm{case} \triangleright (\lambda var_{\,poly} : sig_{\,poly}^{+}.[\mathrm{it} \triangleright exp_p^{\mathrm{case}}])\Big] \\
\Big]
\end{array}
$$

8.6 Pattern Compilation

Patterns $\boxed{\Gamma \vdash pat \Leftarrow exp : con \mid exp' \rightsquigarrow [sdecs] : [sbnds]}$

The judgment should be read as "the bindings of the result of matching *exp* to the pattern *pat* are $[sdecs] : [sbnds]$; if the pattern match fails for any reason, then *exp′* is evaluated." In practice, *exp′* is either fail or raise $basis.\overline{\text{Bind}}$.

$$\frac{\Gamma \vdash_{\text{ctx}} \overline{id} \not\rightsquigarrow \quad \text{or} \quad \Gamma \vdash_{\text{ctx}} \overline{id} \rightsquigarrow path : con' \qquad var \notin \text{bound}(\Gamma)}{\Gamma \vdash id \Leftarrow exp : con \mid exp' \rightsquigarrow [\overline{id} \triangleright var{=}exp] : [\overline{id} \triangleright var{:}con]} \tag{284}$$

Rule 284: Pattern match against an identifier that is not a constructor. (The premise is equivalent to "rules 289 and 291 do not apply".)

$$\frac{lbl \text{ fresh} \qquad var \notin \text{bound}(\Gamma) \qquad \text{type}(scon) = con \qquad \Gamma \vdash exp' : \mathsf{Unit}}{\Gamma \vdash scon \Leftarrow exp : con \mid exp' \rightsquigarrow [lbl \triangleright var{=}\mathsf{if}\ exp =_{con} scon\ \mathsf{then}\ \{\}\ \mathsf{else}\ exp'] : [lbl \triangleright var{:}\mathsf{Unit}]} \tag{285}$$

Rule 285: Pattern match against a constant. We need primitive equality functions for constants which can appear in patterns.

$$\frac{lbl \text{ fresh} \qquad var \notin \text{bound}(\Gamma)}{\Gamma \vdash \_ \Leftarrow exp : con \mid exp' \rightsquigarrow [lbl \triangleright var{=}exp] : [lbl \triangleright var{:}con]} \tag{286}$$

Rule 286: Pattern match against a wildcard.

$$\frac{\Gamma \vdash ty \rightsquigarrow con' : \Omega \qquad \Gamma \vdash con \equiv con' : \Omega \qquad \Gamma \vdash pat \Leftarrow exp : con \mid exp' \rightsquigarrow mod : sig}{\Gamma \vdash pat\ :\ ty \Leftarrow exp : con \mid exp' \rightsquigarrow mod : sig} \tag{287}$$

Rule 287: Pattern match against an explicitly-typed pattern.

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{ctx}} \overline{longid} \rightsquigarrow path : [\mathrm{mk} \triangleright con' \rightarrow con'', \mathrm{km} \triangleright con'' \rightharpoonup con'] \\ \Gamma \vdash con \equiv con'' : \Omega \qquad \Gamma \vdash exp' : con' \\ \Gamma \vdash pat \Leftarrow \mathsf{catch}\,(path.\mathrm{km}\; exp)\,\mathsf{with}\; exp' : con' \mid exp' \rightsquigarrow mod : sig\end{array}}{\Gamma \vdash exp' \Leftarrow exp : con \mid longid\ pat \rightsquigarrow mod : sig} \tag{288}$$

Rule 288: Pattern match against a monomorphic (datatype or exception) constructor which carries a value.

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{ctx}} \overline{longid} \rightsquigarrow path : [\mathrm{mk} \triangleright con_{\mathrm{mk}}, \mathrm{km} \triangleright con'' \rightharpoonup \mathsf{Unit}] \\ \Gamma \vdash con \equiv con'' : \Omega \qquad \Gamma \vdash exp' : \mathsf{Unit} \\ lbl \text{ fresh} \qquad var \notin \mathrm{bound}(\Gamma)\end{array}}{\begin{array}{l}\Gamma \vdash longid \Leftarrow exp : con \mid exp' \rightsquigarrow \\ \qquad [lbl \triangleright var = \mathsf{catch}\; path.\mathrm{km}\; exp\; \mathsf{with}\; exp'] : [lbl \triangleright var : \mathsf{Unit}]\end{array}} \tag{289}$$

Rule 289: Pattern match against a constant, monomorphic (datatype or exception) constructor.

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{ctx}} \overline{longid} \rightsquigarrow path : [\mathrm{mk} \triangleright sig_{\mathrm{mk}}, \mathrm{km} \triangleright sig_{\mathrm{km}}] \\ \Gamma \vdash sig_{\mathrm{km}} \leq (sig' \rightharpoonup [\mathrm{it} \triangleright con' \rightharpoonup con'']) : \mathsf{Sig} \\ \Gamma \vdash sig_{\mathrm{mk}} \leq (sig' \rightarrow [\mathrm{it} \triangleright con'' \rightharpoonup con']) : \mathsf{Sig} \\ \Gamma \vdash con \equiv con'' : \Omega \\ \Gamma \vdash exp' : con'' \\ \Gamma \vdash var' \downarrow sig' \\ \Gamma \vdash pat \Leftarrow \mathsf{catch}\,((path.\mathrm{km}\; var').\mathrm{it})\; exp\; \mathsf{with}\; exp' \mid exp' \rightsquigarrow mod : sig\end{array}}{\Gamma \vdash longid\ pat \Leftarrow exp : con \mid exp' \rightsquigarrow mod : sig} \tag{290}$$

Rule 290: Pattern match against a polymorphic datatype constructor carrying a value.

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{ctx}} \overline{longid} \rightsquigarrow path : [\mathrm{mk} \triangleright sig_{\mathrm{mk}}, \mathrm{km} \triangleright sig_{\mathrm{km}}] \\ \Gamma \vdash sig_{\mathrm{km}} \leq (sig' \rightharpoonup [\mathrm{it} \triangleright con'' \rightharpoonup \mathsf{Unit}]) : \mathsf{Sig} \\ \Gamma \vdash sig_{\mathrm{mk}} \leq (sig' \rightarrow [\mathrm{it} \triangleright con'']) : \mathsf{Sig} \\ \Gamma \vdash con \equiv con'' : \Omega \\ \Gamma \vdash exp' : \mathsf{Unit} \\ \Gamma \vdash var' \downarrow sig'\end{array}}{\begin{array}{l}\Gamma \vdash longid \Leftarrow exp : con \mid exp' \rightsquigarrow \\ \quad [lbl \triangleright var{:}\mathsf{catch}\,((path.\mathrm{km}\ var').\mathrm{it})\ exp\ \mathsf{with}\ exp'] : \\ \quad [lbl \triangleright var{:}\mathsf{Unit}]\end{array}} \tag{291}$$

Rule 291: Pattern match against a constant polymorphic constructor.

$$\frac{\begin{array}{c}lbl\ \text{fresh} \qquad var \notin \mathrm{bound}(\Gamma) \\ \Gamma \vdash con \equiv \{\overline{lab_1} \sim con_1, \cdots, \overline{lab_n} \sim con_n, \\ \qquad \langle lbl'_1 \sim con'_1, \cdots, lbl'_k \sim con'_k \rangle\} : \Omega \\ \Gamma, lbl \triangleright var{:}con \vdash pat_1 \Leftarrow var.\overline{lab_1} : con_1 \mid exp' \rightsquigarrow [sbnds_1] : [sdecs_1] \\ \vdots \\ \Gamma, lbl \triangleright var{:}con \vdash pat_n \Leftarrow var.\overline{lab_n} : con_n \mid exp' \rightsquigarrow [sbnds_n] : [sdecs_n]\end{array}}{\begin{array}{l}\Gamma \vdash \{lab_1 = pat_1, \ldots, lab_n = pat_n \langle, \ldots \rangle\} \Leftarrow exp : con \mid exp' \rightsquigarrow \\ \quad [lbl \triangleright var{=}exp, sbnds_1, \ldots, sbnds_n] : \\ \quad [lbl \triangleright var{:}exp, sdecs_1, \ldots, sdecs_n]\end{array}} \tag{292}$$

Rule 292: Pattern match against a record of patterns.

$$\frac{\begin{array}{c}lbl\ \text{fresh} \qquad var \notin \mathrm{bound}(\Gamma) \\ \Gamma \vdash pat_1 \Leftarrow var : con \mid exp' \rightsquigarrow [sbnds_1] : [sdecs_1] \\ \Gamma \vdash pat_2 \Leftarrow var : con \mid exp' \rightsquigarrow [sbnds_2] : [sdecs_2]\end{array}}{\begin{array}{l}\Gamma \vdash pat_1\ \mathtt{as}\ pat_2 \Leftarrow exp : con \mid exp' \rightsquigarrow \\ \quad [lbl \triangleright var{=}exp, sbnds_1, sbnds_2] : [lbl \triangleright var{:}exp, sdecs_1, sdecs_2]\end{array}} \tag{293}$$

Rule 293: Pattern match against two patterns simultaneously.

$$\frac{\Gamma \vdash pat \Leftarrow \mathsf{get}\ exp : con \mid exp' \rightsquigarrow mod : sig}{\Gamma \vdash \mathtt{ref}\ pat \Leftarrow exp : con\ \mathsf{Ref} \mid exp' \rightsquigarrow mod : sig} \tag{294}$$

Rule 294: Pattern match involving implicit dereferencing of a `ref` cell.

8.7 Lookup Rules

The lookup rules specify the order in which translation contexts and IL signatures are searched.

- To prevent an explosion of rules, the metavariable Θ is used to denote a type, signature, or kind as appropriate.
- Any IL structure label starred with an asterisk (e.g., lbl^*) is treated specially by the lookup, which checks inside the structure. That is, when looking for an identifier in a context, we also look *inside* starred structure declarations. If the identifier is found inside such a structure, the full path to the identifier through the open structure is returned.
- The lookup of signatures in a context is treated specially by returning the signature rather than the path to the signature.
- Any type or signature returned by a lookup will be valid with respect to the ambient context.

Contexts

$$\boxed{\Gamma \vdash_{\text{ctx}} lbls \rightsquigarrow path/sig : \Theta}$$

Main rules for looking up identifiers in a translation context.

$$\frac{\Gamma \vdash_{\text{ctx}} lbls \rightsquigarrow path \qquad \Gamma \vdash path : con}{\Gamma \vdash_{\text{ctx}} lbls \rightsquigarrow path : con} \tag{295}$$

$$\frac{\Gamma \vdash_{\text{ctx}} lbls \rightsquigarrow path \qquad \Gamma \vdash path : knd}{\Gamma \vdash_{\text{ctx}} lbls \rightsquigarrow path : knd} \tag{296}$$

$$\frac{\Gamma \vdash_{\text{ctx}} lbls \rightsquigarrow path \qquad \Gamma \vdash path : sig}{\Gamma \vdash_{\text{ctx}} lbls \rightsquigarrow path : sig} \tag{297}$$

$$\frac{\Gamma \vdash_{\text{ctx}} lbls \rightsquigarrow sig \qquad \Gamma \vdash sig : \mathsf{Sig}}{\Gamma \vdash_{\text{ctx}} lbls \rightsquigarrow sig : \mathsf{Sig}} \tag{298}$$

$$\boxed{\Gamma \vdash_{\text{ctx}} lbls \rightsquigarrow path/sig}$$

"Utility" rules used to look up identifiers in a translation context.

$$\frac{lbl = lbl'}{\Gamma, lbl \triangleright var{:}con \vdash_{\text{ctx}} lbl' \rightsquigarrow var'} \tag{299}$$

$$\frac{lbl \neq lbl' \qquad \Gamma \vdash_{\text{ctx}} lbl' \rightsquigarrow path}{\Gamma, lbl \triangleright var{:}con \vdash_{\text{ctx}} lbl \rightsquigarrow path} \tag{300}$$

$$\frac{lbl = lbl'}{\Gamma, lbl \triangleright var{:}knd\langle{=}con\rangle \vdash_{\text{ctx}} lbl' \rightsquigarrow var} \tag{301}$$

$$\frac{lbl \neq lbl' \qquad \Gamma \vdash_{\text{ctx}} lbl' \rightsquigarrow path}{\Gamma, lbl \triangleright var{:}knd\langle{=}con\rangle \vdash_{\text{ctx}} lbl' \rightsquigarrow path} \tag{302}$$

$$\frac{lbl = lbl'}{\Gamma, lbl \triangleright var{:}sig \vdash_{\text{ctx}} lbl' \rightsquigarrow var} \tag{303}$$

$$\frac{lbl \neq lbl' \qquad \Gamma \vdash_{\text{ctx}} lbl' \rightsquigarrow path}{\Gamma, lbl \triangleright var{:}sig \vdash_{\text{ctx}} lbl' \rightsquigarrow path} \tag{304}$$

$$\frac{sig \vdash_{\text{sig}} lbl' \rightsquigarrow path}{\Gamma, lbl^* \triangleright var{:}sig \vdash_{\text{ctx}} lbl' \rightsquigarrow var.path} \tag{305}$$

$$\frac{sig \vdash_{\text{sig}} lbl' \not\rightsquigarrow \qquad \Gamma \vdash_{\text{ctx}} lbl' \rightsquigarrow path}{\Gamma, lbl^* \triangleright var{:}sig \vdash_{\text{ctx}} lbl' \rightsquigarrow path} \tag{306}$$

$$\frac{lbl = lbl'}{\Gamma, lbl \triangleright var{:}\mathsf{Sig}{=}sig \vdash_{\text{ctx}} lbl' \rightsquigarrow sig} \tag{307}$$

$$\frac{lbl \neq lbl' \qquad \Gamma \vdash_{\text{ctx}} lbl' \rightsquigarrow path}{\Gamma, lbl \triangleright var{:}\mathsf{Sig}{=}sig \vdash_{\text{ctx}} lbl' \rightsquigarrow path} \tag{308}$$

$$\frac{\Gamma \vdash_{\text{ctx}} lbl \rightsquigarrow path \qquad sig \vdash_{\text{sig}} lbls \rightsquigarrow lbls'}{\Gamma \vdash_{\text{ctx}} lbl.lbls \rightsquigarrow path.lbls'} \tag{309}$$

Signatures

$$\boxed{\Gamma, path{:}sig \vdash_{\text{sig}} lbls \rightsquigarrow lbls' : \Theta}$$

Main rules for looking up identifiers in a signature.

$$\frac{\Gamma \vdash path : sig \qquad sig \vdash_{\text{sig}} lbls \rightsquigarrow lbls' \qquad \Gamma \vdash path.lbls' : con}{\Gamma, path{:}sig \vdash_{\text{sig}} lbls \rightsquigarrow lbls' : con} \tag{310}$$

$$\frac{\Gamma \vdash path : sig \qquad sig \vdash_{\text{sig}} lbls \rightsquigarrow lbls' \qquad \Gamma \vdash path.lbls' : sig'}{\Gamma, path{:}sig \vdash_{\text{sig}} lbls \rightsquigarrow lbls' : sig'} \tag{311}$$

$$\frac{\Gamma \vdash path : sig \qquad sig \vdash_{\text{sig}} lbls \rightsquigarrow lbls' \qquad \Gamma \vdash path.lbls' : knd}{\Gamma, path{:}sig \vdash_{\text{sig}} lbls \rightsquigarrow lbls' : knd} \tag{312}$$

$$\boxed{sig \vdash_{\text{sig}} lbls \rightsquigarrow lbls'}$$

"Utility" rules used to look up identifiers in a signature.

$$\frac{lbl = lbl'}{[sdecs, lbl \triangleright var{:}con] \vdash_{\text{sig}} lbl' \rightsquigarrow lbl} \tag{313}$$

$$\frac{lbl \neq lbl' \qquad [sdecs] \vdash_{\text{sig}} lbl' \rightsquigarrow lbls}{[sdecs, lbl \triangleright var{:}con] \vdash_{\text{sig}} lbl' \rightsquigarrow lbls} \tag{314}$$

$$\frac{lbl = lbl'}{[sdecs, lbl \triangleright var{:}knd\langle{=}con\rangle] \vdash_{\text{sig}} lbl' \rightsquigarrow lbl} \tag{315}$$

$$\frac{lbl \neq lbl' \qquad [sdecs] \vdash_{\text{sig}} lbl' \rightsquigarrow lbls}{[sdecs, lbl \triangleright var{:}knd\langle{=}con\rangle] \vdash_{\text{sig}} lbl' \rightsquigarrow lbls} \tag{316}$$

$$\frac{lbl = lbl'}{[sdecs, lbl \triangleright var{:}sig] \vdash_{\text{sig}} lbl' \rightsquigarrow lbl} \tag{317}$$

$$\frac{lbl \neq lbl' \qquad [sdecs] \vdash_{\text{sig}} lbl' \rightsquigarrow lbls}{[sdecs, lbl \triangleright var{:}sig] \vdash_{\text{sig}} lbl' \rightsquigarrow lbls} \tag{318}$$

$$\frac{sig \vdash_{\text{sig}} lbl' \rightsquigarrow lbls}{[sdecs, lbl^* \triangleright var{:}sig] \vdash_{\text{sig}} lbl' \rightsquigarrow lbl^*.lbls} \tag{319}$$

$$\frac{sig \vdash_{\text{sig}} lbl' \not\rightsquigarrow \qquad [sdecs] \vdash_{\text{sig}} lbl' \rightsquigarrow lbls}{[sdecs, lbl^* \triangleright var{:}sig] \vdash_{\text{sig}} lbl' \rightsquigarrow lbls} \tag{320}$$

$$\frac{sig \vdash_{\text{ctx}} lbl \rightsquigarrow lbls' : sig' \qquad sig' \vdash_{\text{sig}} path \rightsquigarrow lbls''}{sig \vdash_{\text{sig}} lbl.lbls \rightsquigarrow lbls'.lbls''} \tag{321}$$

8.8 Coercions

Matching Coercions

$$\boxed{\begin{array}{l} decs \vdash_{\text{sub}} sig_0 \preceq sig \rightsquigarrow \\ \qquad (\lambda var_0{:}sig_0.mod) : \\ \qquad ((var_0{:}sig_0) \rightarrow sig') \end{array}}$$

This judgment should be read as "the coercion functor that copies the components appearing in sig from a structure matching signature sig_0 is $(\lambda(var_0{:}sig_0).mod)$ and has signature $((var_0{:}sig_0) \rightarrow sig')$; this signature maximizes type propagation."

$$\frac{}{decs \vdash_{\text{sub}} sig_0 \preceq [] \rightsquigarrow (\lambda var_0{:}sig_0.[]) : ((var_0{:}sig_0) \rightarrow [])} \tag{322}$$

Rule 322: Coercion to forget all components.

$$\frac{\begin{array}{c} decs, var_0{:}sig_0 \vdash_{\text{sig}} lbl \rightsquigarrow lbls : con' \\ decs \vdash con \equiv con' : \Omega \\ decs, var{:}con \vdash_{\text{sub}} sig_0 \preceq [sdecs] \rightsquigarrow \\ \qquad (\lambda var_0{:}sig_0.[sbnds]) : ((var_0{:}sig_0) \rightarrow [sdecs']) \end{array}}{\begin{array}{l} decs \vdash_{\text{sub}} sig_0 \preceq [lbl \triangleright var{:}con, sdecs] \rightsquigarrow \\ \qquad (\lambda var_0{:}sig_0.[lbl \triangleright var{=}var_0.lbls, sbnds]) : \\ \qquad ((var_0{:}sig_0) \rightarrow [lbl \triangleright var{:}con, sdecs']) \end{array}} \tag{323}$$

Rule 323: Coercion of a monomorphic value specification to a monomorphic value specification.

$$\frac{\left\{\begin{array}{c} decs, var_0{:}sig_0 \vdash_{\text{sig}} lbl \rightsquigarrow lbls : sig \\ \text{or} \\ decs, var_0{:}sig_0 \vdash_{\text{sig}} lbl \rightsquigarrow lbls : [\text{mk}\triangleright sig, \text{km}\triangleright sig_{\text{km}}] \end{array}\right\} \\ decs \vdash sig \leq (sig_{poly}{\rightarrow}[\text{it}\triangleright con]) : \mathsf{Sig} \\ decs, var_1{:}sig_1 \vdash var_{poly} \downarrow sig_{poly} \\ decs, var{:}((var_1{:}sig_1){\rightarrow}[\text{it}\triangleright con]) \vdash_{\text{sub}} sig_0 \preceq [sdecs] \rightsquigarrow \\ (\lambda var_0{:}sig_0.[sbnds]) : ((var_0{:}sig_0){\rightarrow}[sdecs'])}{\begin{array}{l} decs \vdash_{\text{sub}} sig_0 \preceq [lbl\triangleright var{:}((var_1{:}sig_1){\rightarrow}[\text{it}\triangleright con]), sdecs] \rightsquigarrow \\ \quad (\lambda var_0{:}sig_0.[lbl\triangleright var{=}(\lambda var_1{:}sig_1.\left\{\begin{array}{c} var_0.lbls \\ \text{or} \\ var_0.lbls.\text{mk} \end{array}\right\} var_{poly}), sbnds]) : \\ \quad ((var_0{:}sig_0){\rightarrow}[lbl\triangleright var{:}((var_1{:}sig_1){\rightarrow}[\text{it}\triangleright con]), sdecs']) \end{array}} \quad (324)$$

Rule 324: Coercion of a polymorphic value specification to a polymorphic value specification; this may involve implicit polymorphic instantiation. (The rule also handles matching polymorphic datatype and exception constructors with value specifications.)

$$\frac{\left\{\begin{array}{c} decs, var_0{:}sig_0 \vdash_{\text{sig}} lbl \rightsquigarrow lbls : sig \\ \text{or} \\ decs, var_0{:}sig_0 \vdash_{\text{sig}} lbl \rightsquigarrow lbls : [\text{mk}\triangleright sig, \text{km} \sim sig_{\text{km}}] \end{array}\right\} \\ decs \vdash sig \leq (sig_{poly}{\rightarrow}[\text{it}\triangleright con']) : \mathsf{Sig} \\ decs \vdash con \equiv con' : \Omega \\ decs, var_1{:}sig_1 \vdash var_{poly} \downarrow sig_{poly} \\ decs, var{:}con \vdash_{\text{sub}} sig_0 \preceq [sdecs] \rightsquigarrow \\ (\lambda var_0{:}sig_0.[sbnds]) : ((var_0{:}sig_0){\rightarrow}[sdecs'])}{\begin{array}{l} decs \vdash_{\text{sub}} sig_0 \preceq [lbl\triangleright var{:}con, sdecs] \rightsquigarrow \\ \quad (\lambda var_0{:}sig_0.[lbl\triangleright var{=}(\lambda var_1{:}sig_1.(\left\{\begin{array}{c} var_0.lbls \\ \text{or} \\ var_0.lbls.\text{mk} \end{array}\right\} var_{poly}).\text{it}), sbnds]) : \\ \quad ((var_0{:}sig_0){\rightarrow}[lbl\triangleright var{:}con, sdecs']) \end{array}} \quad (325)$$

Rule 325: Coercion of a polymorphic value (or datatype constructor) to match a monomorphic value specification.

$$\frac{\begin{array}{c}decs, var_0{:}sig_0 \vdash_{\text{sig}} lbl \rightsquigarrow lbls : knd \\ \langle decs, var_0{:}sig_0 \vdash var_0.lbls \equiv con : knd \rangle \\ decs, var{:}knd{=}var_0.lbls \vdash_{\text{sub}} sig_0 \preceq [sdecs] \rightsquigarrow \\ (\lambda var_0{:}sig_0.[sbnds]) : ((var_0{:}sig_0){\rightarrow}[sdecs'])\end{array}}{\begin{array}{l}decs \vdash_{\text{sub}} sig_0 \preceq [lbl \triangleright var{:}knd\langle{=}con\rangle, sdecs] \rightsquigarrow \\ \quad (\lambda var_0{:}sig_0.[lbl \triangleright var{=}var_0.lbls, sbnds]) : \\ \quad ((var_0{:}sig_0){\rightarrow}[lbl \triangleright var{:}knd{=}var_0.lbls, sdecs'])\end{array}} \tag{326}$$

Rule 326: Coercion of a type component to a type component exposing less or equal information.

$$\frac{\begin{array}{c}decs, var_0{:}sig_0 \vdash_{\text{sig}} lbl \rightsquigarrow lbls : sig_1 \\ decs \vdash_{\text{sub}} sig_1 \preceq sig \rightsquigarrow (\lambda var_1{:}sig_1.mod) : ((var_1{:}sig_1){\rightarrow}sig'') \\ decs, var{:}sig \vdash_{\text{sub}} sig_0 \preceq [sdecs] \rightsquigarrow \\ (\lambda var_0{:}sig_0.[sbnds]) : ((var_0{:}sig_0){\rightarrow}[sdecs'])\end{array}}{\begin{array}{l}decs \vdash_{\text{sub}} sig_0 \preceq [lbl \triangleright var{:}sig, sdecs] \rightsquigarrow \\ \quad (\lambda var_0{:}sig_0.[lbl \triangleright var{=}(\lambda var_1{:}sig_1.mod)\, var_0.lbls, sbnds]) : \\ \quad ((var_0{:}sig_0){\rightarrow}[lbl \triangleright var{:}sig'', sdecs'])\end{array}} \tag{327}$$

Rule 327: Coercion of a module component.

$$\frac{\begin{array}{c}decs \vdash_{\text{sub}} sig_0 \preceq [sdecs''] \rightsquigarrow (\lambda var_0{:}sig_0.mod) : ((var_0{:}sig_0){\rightarrow}sig') \\ decs, var{:}sig \vdash_{\text{sub}} sig_0 \preceq [sdecs] \rightsquigarrow \\ (\lambda var_0{:}sig_0.[sbnds]) : ((var_0{:}sig_0){\rightarrow}[sdecs'])\end{array}}{\begin{array}{l}decs \vdash_{\text{sub}} sig_0 \preceq [lbl^* \triangleright var{:}[sdecs''], sdecs] \rightsquigarrow \\ \quad (\lambda var_0{:}sig_0.[lbl^* \triangleright var{=}mod, sbnds]) : \\ \quad ((var_0{:}sig_0){\rightarrow}[lbl^* \triangleright var{:}sig', sdecs'])\end{array}} \tag{328}$$

Rule 328: Coercion to a starred substructure—this structure need not actually exist in the original structure, just the components.

$$\frac{\begin{array}{c} decs \vdash_{\text{sub}} sig_2 \preceq sig_1 \rightsquigarrow mod_3 : sig_3 \\ decs \vdash_{\text{sub}} sig'_1 \preceq sig'_2 \rightsquigarrow mod_4 : sig_4 \\ decs, var_0{:}(var_1{:}sig_1 \rightharpoonup sig'_1), var_2{:}sig_2 \vdash mod_4\,(var_0\,(mod_3\,var_2)) : sig \end{array}}{\begin{array}{l} decs \vdash_{\text{sub}} (var_1{:}sig_1 \rightharpoonup sig'_1) \preceq (var_2{:}sig_2 \rightharpoonup sig'_2) \rightsquigarrow \\ \quad (\lambda var_0{:}(var_1{:}sig_1 \rightharpoonup sig'_1).(\lambda var_2{:}sig_2.mod_4\,(var_0\,(mod_3\,var_2)))) : \\ \quad ((var_0{:}(var_1{:}sig_1 \rightharpoonup sig'_1)) \rightarrow (var_2{:}sig_2 \rightharpoonup sig)) \end{array}} \tag{329}$$

$$\frac{\begin{array}{c} decs \vdash_{\text{sub}} sig_2 \preceq sig_1 \rightsquigarrow mod_3 : sig_3 \\ decs \vdash_{\text{sub}} sig'_1 \preceq sig'_2 \rightsquigarrow mod_4 : sig_4 \\ decs, var_0{:}(var_1{:}sig_1 \rightarrow sig'_1), var_2{:}sig_2 \vdash mod_4\,(var_0\,(mod_3\,var_2)) : sig \end{array}}{\begin{array}{l} decs \vdash_{\text{sub}} (var_1{:}sig_1 \rightarrow sig'_1) \preceq (var_2{:}sig_2 \rightarrow sig'_2) \rightsquigarrow \\ \quad (\lambda var_0{:}(var_1{:}sig_1 \rightarrow sig'_1).(\lambda var_2{:}sig_2.mod_4\,(var_0\,(mod_3\,var_2)))) : \\ \quad ((var_0{:}(var_1{:}sig_1 \rightarrow sig'_1)) \rightarrow (var_2{:}sig_2 \rightarrow sig)) \end{array}} \tag{330}$$

Rules 329, 330: Coercions for functor subtyping. By the IL subtyping rules, we can use Rule 329 to coerce a total functor to a partial functor.

8.9 Signature Patching

where type

$$\boxed{sig \vdash_{\text{wt}} lbls := con : knd \rightsquigarrow sig' : \mathsf{Sig}}$$

This judgment should be read "By adding to the signature sig the fact that the (abstract) type component selected by $lbls$ is equal to $con : knd$, we get the signature sig'."

$$\frac{\begin{array}{c}\mathrm{FV}(con) \cap \mathrm{bound}(sdecs) = \emptyset \\ sig = [sdecs, lbl \triangleright var{:}knd, sdecs']\end{array}}{sig \vdash_{\text{wt}} lbl := con : knd \rightsquigarrow [sdecs, lbl \triangleright var{:}knd{=}con, sdecs'] : \mathsf{Sig}} \tag{331}$$

$$\frac{\begin{array}{c}\mathrm{FV}(con) \cap \mathrm{bound}(sdecs) = \emptyset \\ sig = [sdecs, lbl \triangleright var{:}sig', sdecs'] \\ sig' \vdash_{\text{wt}} lbls := con : knd \rightsquigarrow sig'' : \mathsf{Sig}\end{array}}{sig \vdash_{\text{wt}} lbl.lbls := con : knd \rightsquigarrow [sdecs, lbl \triangleright var{:}sig'', sdecs'] : \mathsf{Sig}} \tag{332}$$

sharing

$$\boxed{sig \vdash_{\text{sh}} lbls_1 := lbls_2 : knd \rightsquigarrow sig' : \mathsf{Sig}}$$

This judgment should be read "By adding to the signature sig the fact that the (abstract) type components of kind knd selected by $lbls_1$ and $lbls_2$ are equal, we get the signature sig'."

$$\frac{sig = [sdecs, lbl' \triangleright var'{:}knd, sdecs', lbl \triangleright var{:}knd, sdecs'']}{\begin{array}{l}sig \vdash_{\text{sh}} lbl := lbl' : knd \rightsquigarrow \\ \quad [sdecs, lbl' \triangleright var'{:}knd, sdecs', lbl \triangleright var{:}knd{=}var', sdecs'] : \mathsf{Sig}\end{array}} \tag{333}$$

$$\frac{\begin{array}{c}sig = [sdecs, lbl' \triangleright var'{:}knd, sdecs', lbl \triangleright var{:}sig, sdecs''] \\ sig \vdash_{\text{wt}} lbls := var' : knd \rightsquigarrow sig' : \mathsf{Sig}\end{array}}{\begin{array}{l}sig \vdash_{\text{sh}} lbl.lbls := lbl' : knd \rightsquigarrow \\ \quad [sdecs, lbl' \triangleright var'{:}knd, sdecs', lbl \triangleright var{:}sig', sdecs'] : \mathsf{Sig}\end{array}} \tag{334}$$

$$\frac{\begin{array}{c}sig = [sdecs, lbl' \triangleright var'{:}sig', sdecs', lbl \triangleright var{:}sig'', sdecs''] \\ sig'' \vdash_{\text{wt}} lbls' := var'.lbls' : knd \rightsquigarrow sig''' : \mathsf{Sig}\end{array}}{\begin{array}{l}sig \vdash_{\text{sh}} lbl.lbls := lbl'.lbls' : knd \rightsquigarrow \\ \quad [sdecs, lbl' \triangleright var'{:}sig', sdecs', lbl \triangleright var{:}sig''', sdecs''] : \mathsf{Sig}\end{array}} \tag{335}$$

$$\frac{\begin{array}{c} sig = [sdecs, lbl \triangleright var{:}sig', sdecs'] \\ sig' \vdash_{\mathrm{sh}} lbls := lbls' : knd \rightsquigarrow sig'' : \mathsf{Sig} \end{array}}{sig \vdash_{\mathrm{sh}} lbl.lbls := lbl.lbls' : knd \rightsquigarrow [sdecs, lbl \triangleright var{:}sig'', sdecs'] : \mathsf{Sig}} \quad (336)$$

9 Conjectures

1. **[Type Preservation under Evaluation].**

   If $\vdash tbnds : tdecs$ and $\cdot, \cdot \vdash tbnds \Downarrow \Delta, \sigma, \mathsf{VALUES}(tbnds')$ then $\Delta \vdash tbnds' : tdecs$.

2. **[Canonical Forms].**

If a value has type/sig...	*then it is of the form...*
$con_1 \rightharpoonup con_2$	$\mathsf{fix}\, fbnds, var'{=}(var{:}con_1){\mapsto}exp, fbnds'\, \mathsf{in}\, var'\, \mathsf{end}$
$con_1 \rightarrow con_2$	$\mathsf{fix}\, fbnds, var'{=}(var{:}con_1){\mapsto}exp, fbnds'\, \mathsf{in}\, var'\, \mathsf{end}$
$\{lbl_1 \triangleright con_1, \cdots, lbl_n \triangleright con_n\}$	$\{lbl_1 \triangleright exp_{\mathrm{v}1}, \cdots, lbl_n \triangleright exp_n\}$
$\Sigma\,(con_1, \cdots, con_n)$	$\mathsf{inj}_i^{\Sigma(con_1,\cdots,con_n)}\, exp_{\mathrm{v}}$
Any	$\mathsf{tag}^{name}\, exp_{\mathrm{v}}$
$con\ \mathsf{Ref}$	loc
base type	$scon$
$[sdecs]$	$[sbnds_{\mathrm{v}}]$
$(var{:}sig \rightharpoonup sig')$	$(\lambda var{:}sig.mod)$
$(var{:}sig \rightarrow sig')$	$(\lambda var{:}sig.mod)$

3. **[Progress under Evaluation].** Well-typed IL programs never get "stuck" during evaluation.

4. **[Determinacy of Evaluation].** If $\cdot, \cdot \vdash tbnds \Downarrow \Delta, \sigma, \mathsf{VALUES}(tbnds)$ and $\cdot, \cdot \vdash tbnds \Downarrow \Delta', \sigma', \mathsf{VALUES}(tbnds')$ then $\Delta, \sigma, tbnds \equiv \Delta', \sigma', tbnds'$ up to renaming of exception names and locations bound in Δ and Δ'.

5. **[Translation Result is Well-Typed].** If $lbl^* \triangleright basis{:}sig_{basis} \vdash topdec \rightsquigarrow tbnds : tdecs$ then $basis{:}sig_{basis} \vdash tbnds : tdecs$.

6. **[Translation Preserves Types].** Under an appropriate correspondence between EL and IL types/signatures/kinds, the translation preserves this.

References

[Har93] Robert Harper. A simplified account of polymorphic references. Technical Report CMU-CS-93-169, School of Computer Science, Carnegie Mellon University, 1993.

[HL94] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *21st Symposium on Principles of Programming Languages*, pages 123–137, 1994.

[HM93] Robert Harper and John C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, 1993.

[HM95] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *22nd Symposium on Principles of Programming Languages*, pages 130–141, 1995.

[HMM90] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *17th Symposium on Principles of Programming Languages*, pages 341–354, 1990.

[Ler94] Xavier Leroy. Manifest types, modules, and separate compilation. In *21st Symposium on Principles of Programming Languages*, pages 109–122, 1994.

[Mor95] Greg Morrisett. *Compiling with Types.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1995.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[Tar96] David Tarditi. *Optimizing ML.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1996. Forthcoming.

[TMC+96] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, 1996. To appear.

[WF91] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. Technical Report TR91-160, Department of Computer Science, Rice University, 1991.